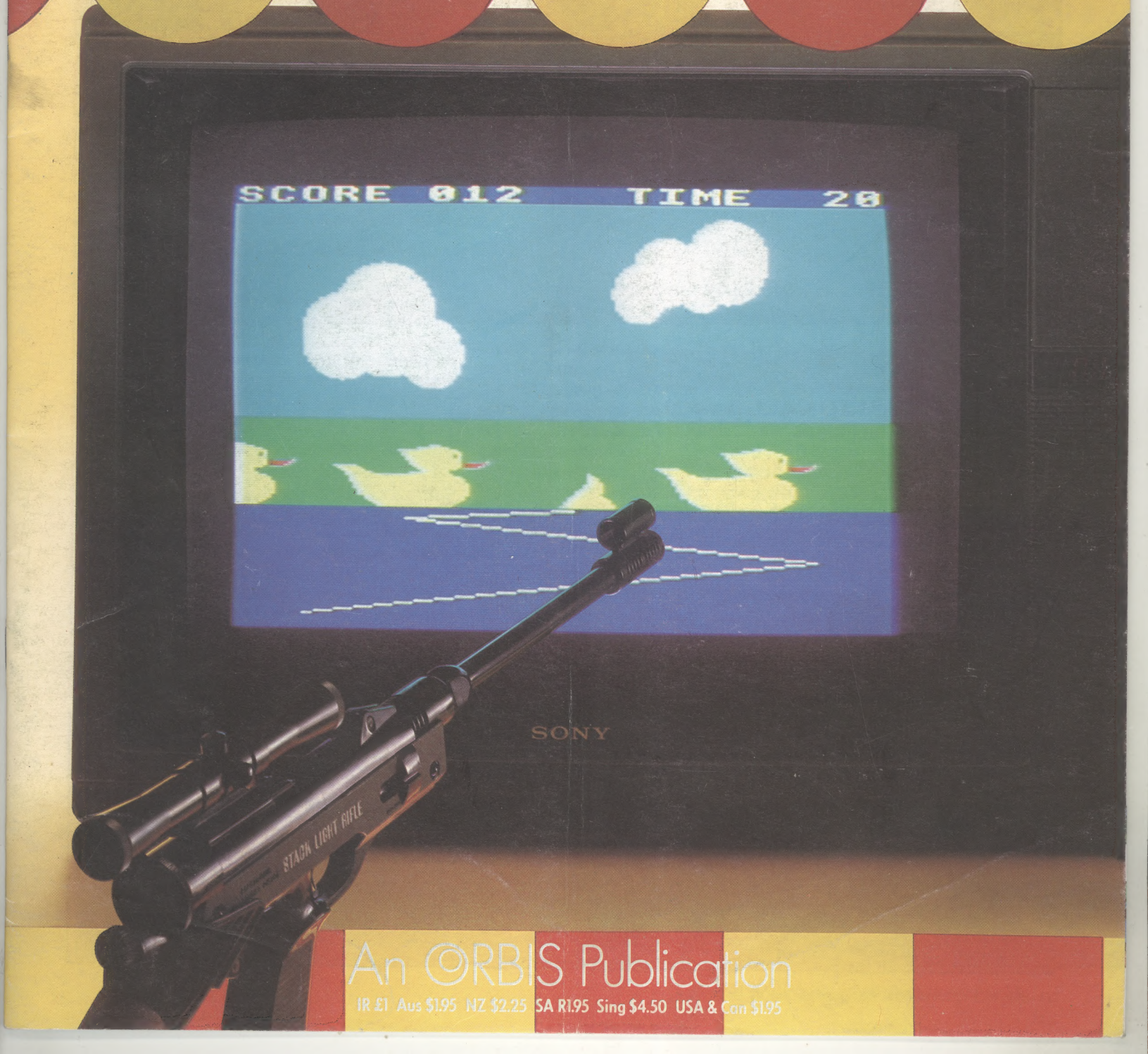
ISSN 0265-2919

80p 12

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

APPLICATION

CONTROLLING INTEREST We look at the fundamental components of a computer-controlled system 221

HARDWARE

RIFLE RANGE The Stack Light Rifle brings authenticity to arcade-style games 230

SOFTWARE

CONTINUING SERIAL We examine the way serial files are created, accessed and updated 226

COMPUTER SCIENCE

SET PIECE A detailed look at the design and function of an R-S flip-flop 228

PROGRAMMING PROJECTS

GRAPHIC DESIGN In this new series we create intricate geometric patterns from simple programs 224

DEFINING TERMS We show you how to define your own sprite graphics on the Commodore 64 232

JARGON

FROM CAD/CAM TO CALL A weekly glossary of computing terms 235

MACHINE CODE

GENERAL ROUTINE By implementing symbols and labels in our programming, we can create portable routines 236

PROFILE

POPULAR CHOICE Digital Research is the company that developed the popular CP/M operating system 239

Next Week

- **We look at the ACT Apricot, a 16-bit business machine hailed as Britain's answer to the IBM PC.**
- **BASICODE is a standard language that can be used by many home computers. We see how well the language overcomes the problems of compatibility.**
- **We continue our look at graphics generation on the Commodore 64.**

QUIZ

1) Suggest a reason why removing the barrel reduces the accuracy of the Stack Light Rifle.

2) 'Feedback' is considered a disadvantage on an audio amplifier. When is it useful to a computer?

3) Why is a NOR gate useful in the construction of flip-flops?

4) Which binary number would you use to set bit 7 in a byte to zero? What is the name of this operation?

Answers To Last Week's Quiz

A1) CLC
LDA ADR1
ADC ADR2
STA ADR3

A2) AND A
LD A, (ADR1)
LD HL, ADR2
ADDA , (HL)
LD (ADR3), A

A3) POKE 53280,0: POKE 53281,1

A4) There can be up to eight different sprites operating at any one time (one for each priority level).

A5) Address, Control and Data.

A6) Originally CP/M stood for Control Program/Monitor, but has become accepted as Control Program for Microprocessors.

QUIZ

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

Editor Max Phillips; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Picture Editor** Claudia Zeff; **Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Contributors** Lisa Kelly, Steven Colwill, Henry Budgett, Jim Lennox, Richard Pawson; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Chris Cooper; **Production Co-ordinator** Ian Paton; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 85 Charlotte Street, London W1P 1LB;

© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derbys.

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

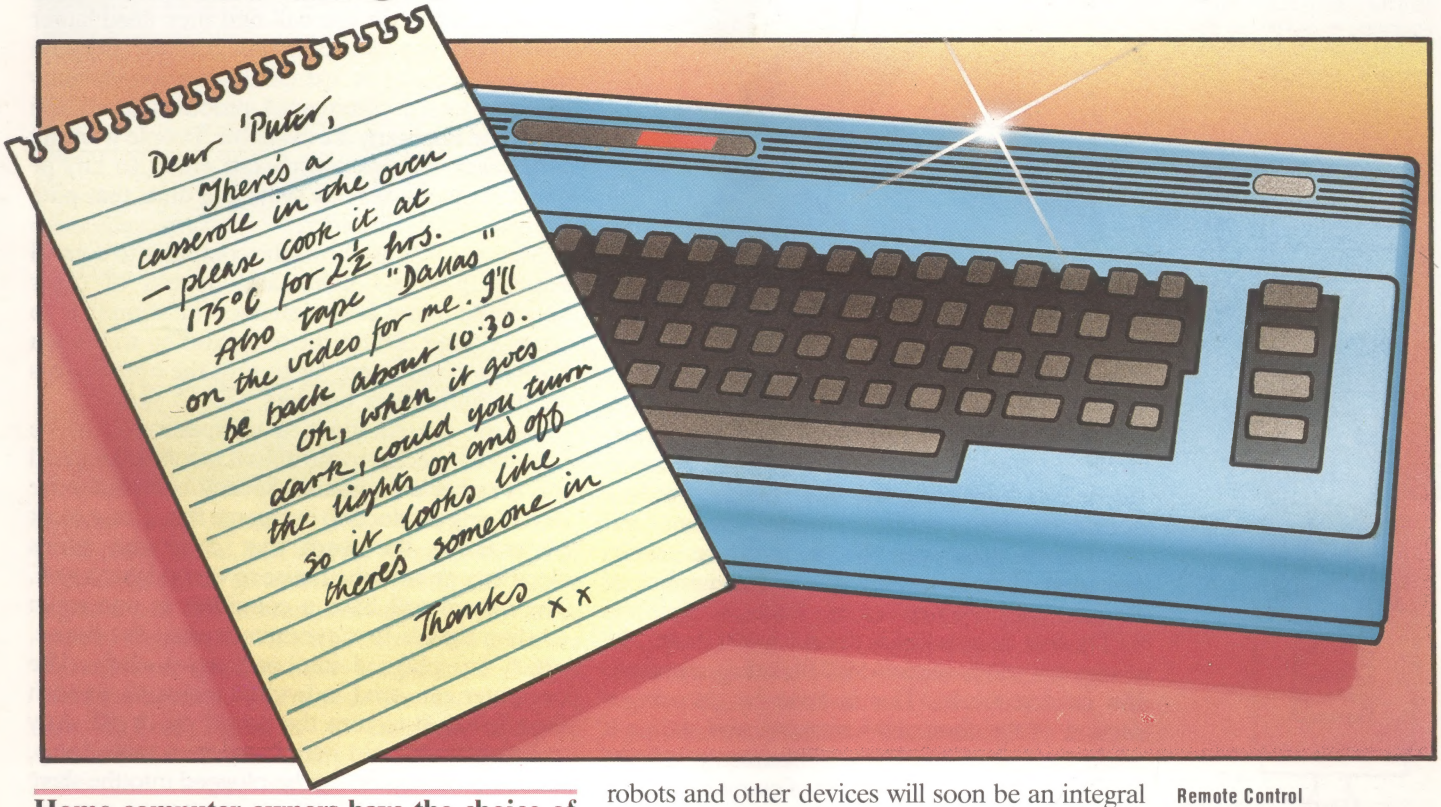
How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

CONTROLLING INTEREST


Dear 'Puter,
There's a casserole in the oven – please cook it at 175°C for 2½ hrs. Also tape "Dallas" on the video for me. I'll be back about 10.30. Oh, when it goes dark, could you turn the lights on and off so it looks like there's someone in
Thanks
x x

MIKE BROWNLOW

Home computer owners have the choice of either buying software or writing their own programs. But few would make the choice between buying or building their own peripheral equipment. Yet there are now components on the market that make it easy to build a computer-controlled device that does exactly what you want it to do.

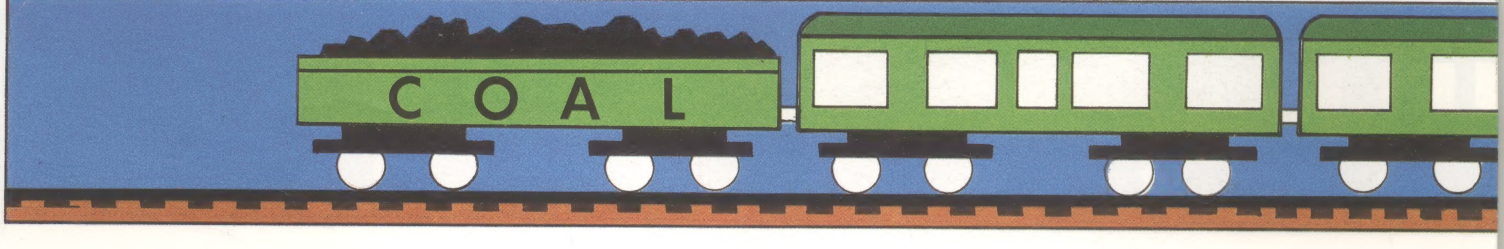
The only reason you really need to set up a micro so that it opens the curtains in the morning or waters the plants while you are on holiday is because it's fun to do. And there's nothing wrong with doing something simply because it's fun. After all, hobbyists spend all those hours writing their own software because they enjoy doing it.

At the present time, building your own peripheral devices may be regarded as 'playing around' with home-made gadgets, but this could prove to be a productive activity in the long run. Many people believe that computer-controlled robots and other devices will soon be an integral part of our lives (in much the same way as computers have become commonplace in as little as five years), and therefore skills learned now could be invaluable in the future. After all, giant computer companies like Apple and Atari began in the back garages of people just 'messing about' with electronic gadgets and components.

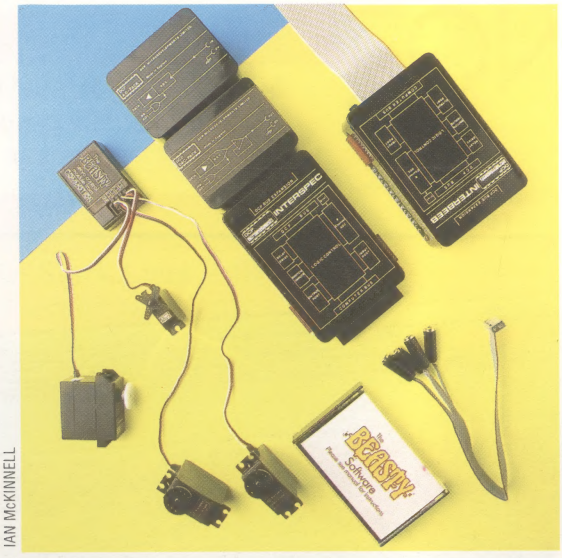
There are a number of elements needed in any computer-controlled system. Obviously there needs to be the computer itself and the item under control. There also needs to be some means for the computer to convey the control messages to the device, and software to enable the computer to decide what those messages should be. Yet this is only half the story. The computer usually needs to have some way of measuring what effect its control is having so that it can make fine adjustments. This is known as *feedback* and without it the computer is as useless as a blindfolded car driver.

All computer-controlled systems rely on control by electric signals. Unfortunately, the tiny

Remote Control
Linking a computer to equipment provides all sorts of possibilities for automatic operation under the control of a program. The computer can respond at pre-programmed times or react to events such as a drop in temperature or a burglar alarm being set off



Control Connection
Interfaces suitable for control uses are available ready-made for use with many home micros, notably the BBC and ZX Spectrum. These units are usually switching units based on relays. The computer can turn the current to particular equipment on and off and can read back whether or not the sensor is 'on' or 'off'



IAN McKINNELL

electric signals used inside a computer are far too small to be of direct use. Even a device as small as a torch bulb uses considerably more power than any of the parts inside a computer; so some means is needed of translating the tiny voltages inside a computer into more powerful ones (and sometimes these can be as great as mains voltages). This is usually done in a number of stages.

The first stage is inside the computer itself. The computer needs some way of sending signals to the outside world. This is done by choosing a section of the computer's memory and setting it aside for just this purpose. The microprocessor will send messages to this part of memory just like any other, but when they get there they are treated differently. This is because this part of memory is known as a *user port* and the information stored in it can be read electronically from outside the micro without affecting its working in any way.

Some micros have a user port as standard, others can have them fitted as an option. By itself the user port can be used to turn LEDS (light-emitting diodes) on and off, but most practical systems need other components as well. Perhaps the most useful is to add a few electronic components, plus a small extra source of electric power, to allow relays to be controlled. *Relays* are essentially switches that can turn on and off relatively large electric currents and yet can be controlled by small electric currents. They provide one of the best ways of converting the small electric pulses used by a computer into useful currents. One of the first aims of anybody experimenting with computer control should be to get a system that allows relays to be used. This is because so many different things can be controlled by relays. Everything from electric motors, water pumps, lights, bells and buzzers to model railways and radio-controlled model cars.

Most relays used by home enthusiasts can cope only with the sort of equipment that will run off batteries. This gives plenty of scope for most projects. A few people will find they need larger relays that can switch mains current. Because mains electricity is extremely dangerous, only properly tested commercial products should be used, and not many are available. People working with small voltages can choose either to buy or make ready-made relay switching units that plug straight into a computer.

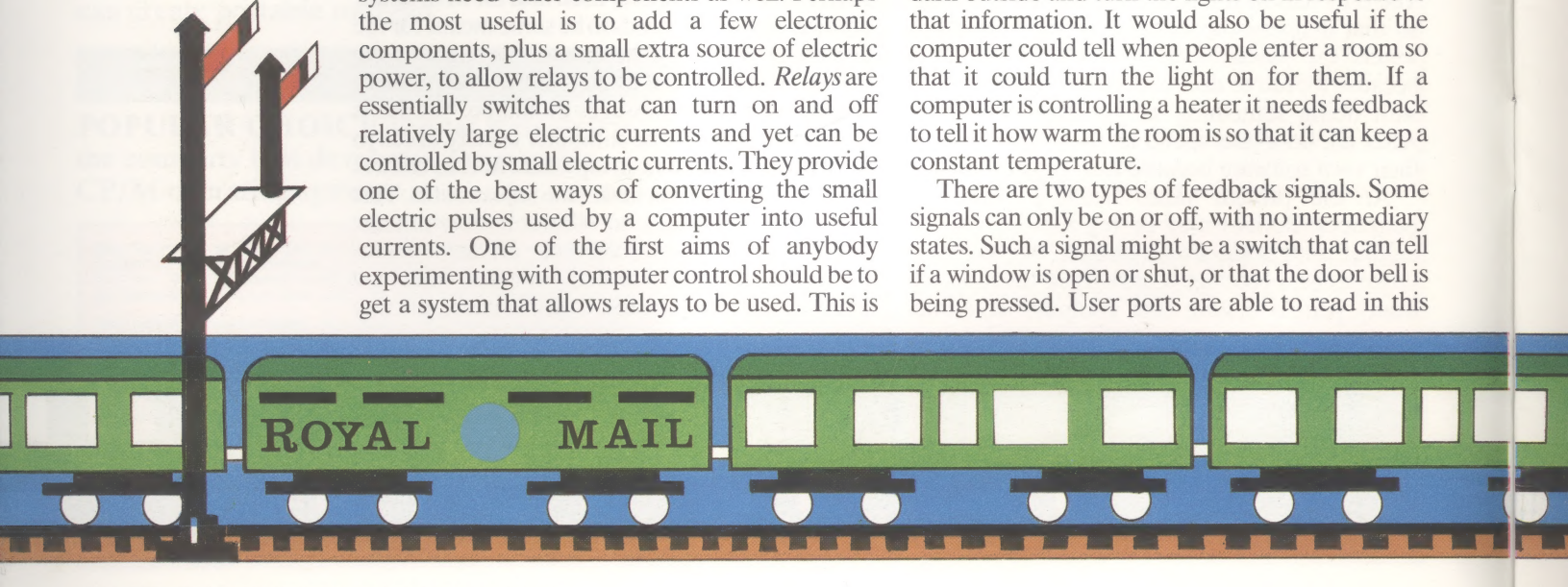
Mains switching allows the computer to control heaters, powerful lights and dozens of other household items. It also enables the computer to act as a timer to turn the television on to catch a favourite programme, or to turn the house lights on and off to deter burglars.

At the present time, a computer has to be directly connected to the units it is controlling, and this is rather a limitation. Several companies are developing products to overcome this. These work by enabling the wires that carry the mains electricity around the house to carry data as well. The system will have a computer in one room sending signals to *slave* units around the house, which are plugged into ordinary sockets. The computer can send individual messages to each slave telling it to turn the power on or off. Any ordinary household item, such as a table lamp, television or heater, can be plugged into the slave and thus controlled by the computer.

FEEDBACK SIGNALS

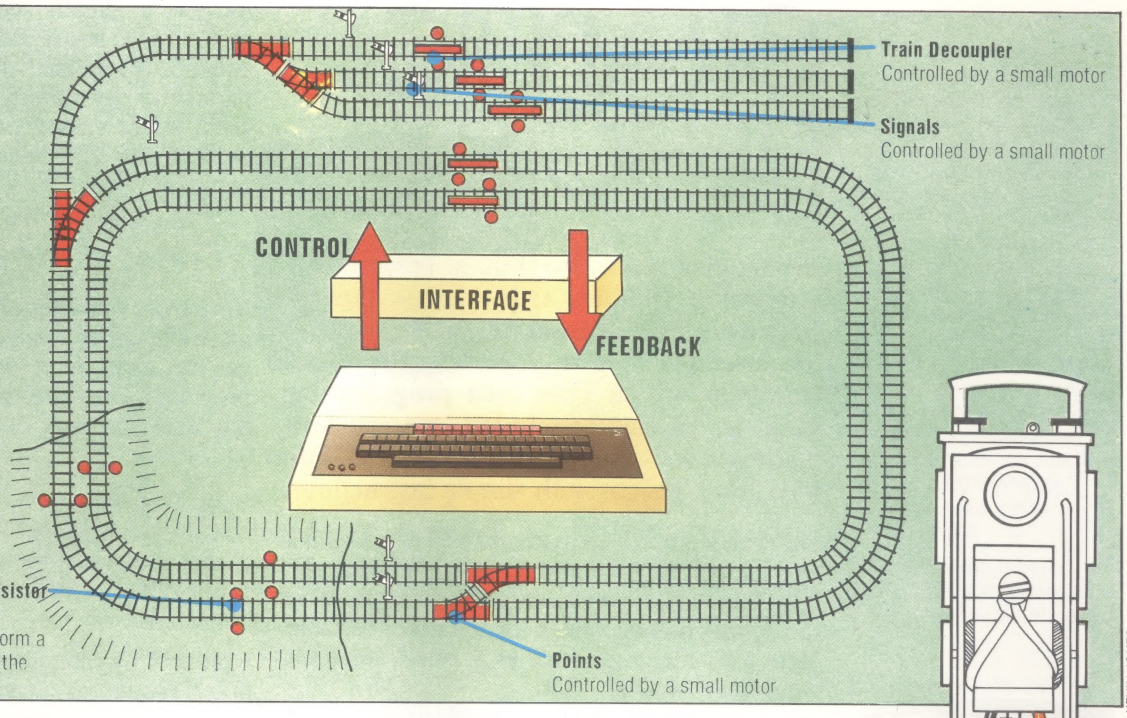
It has already been said that almost all computer-controlled systems need some kind of feedback to measure how the system is working. It might seem that there's no need for feedback when a computer is just turning lights on and off. Yet it would be much better if the computer could tell when it was dark outside and turn the lights on in response to that information. It would also be useful if the computer could tell when people enter a room so that it could turn the light on for them. If a computer is controlling a heater it needs feedback to tell it how warm the room is so that it can keep a constant temperature.

There are two types of feedback signals. Some signals can only be on or off, with no intermediary states. Such a signal might be a switch that can tell if a window is open or shut, or that the door bell is being pressed. User ports are able to read in this



Railway Brain

Controlling any piece of equipment with a micro — from a model railway to an entire house — involves the same basic technique. A loop is established in which the micro sends out control signals, via an interface, to the device concerned. These control servo motors, lights and so on. The device then returns feedback information from sensors such as trip-switches and photo-sensitive cells. This feedback loop allows the equipment to be precisely operated by the computer



KEVIN JONES

type of *on/off signal.*

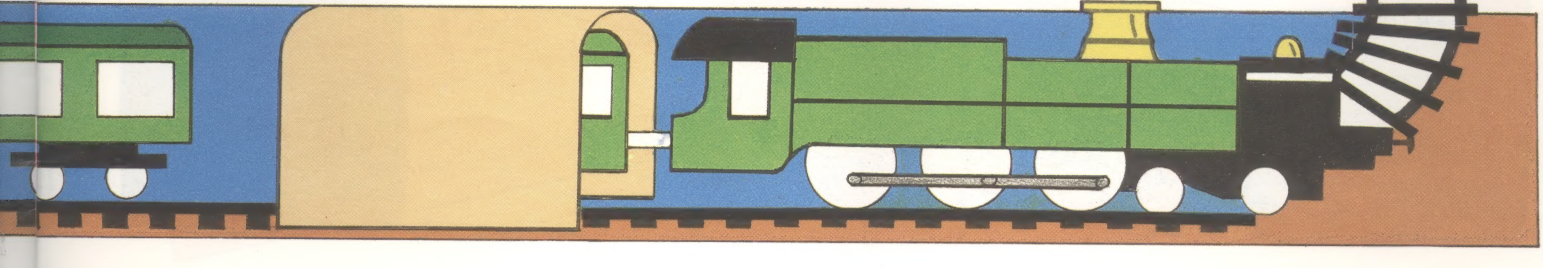
A more useful, but slightly more complicated type of feedback gives an *analogue signal.* Such a signal can be one of a range of values and so can be used to measure how hot it is, how far an object has moved or turned, how heavy something is or what voltage a battery is giving. A device that can accept this sort of signal is an *analogue-to-digital converter* (A/D for short), so-called because it takes the varying range of values from the equipment being controlled (the analogue signal) and converts these into a digital form that the computer can understand.

Having feedback makes a big difference to what can be done under computer control. If a motor is turning a wheel, the computer might be able to estimate how far the wheel has turned in a certain time. This wouldn't work, however, if a load is put on the wheel or the batteries driving the motor are going flat, because the wheel would turn more slowly. An optical sensor could tell the computer each time the wheel completed one revolution, so that the computer could keep track of it.

Some types of electric motor made specially for control uses have a kind of feedback built in to them. This means the computer sends a signal telling them to move to a particular position and the motor keeps working until it has reached it. There are two main types of such motors: *stepper motors* and *servo motors.* A stepper motor can spin continuously, like an ordinary motor, or it can be stopped at any position. However, it lacks power and so can only cope with small loads. Servo motors are powerful but can only turn through a small angle—usually just over 90 degrees. This is often converted into a push/pull type of movement. Both stepper and servo motors need special control units to make them work with computers and these are not available for many of the less popular makes of home computer. Servo motors are used in many robot arms. A number of small robot arms are available that can interface with home computers, but they are very expensive. It is possible to build them from a few servos at lower cost.

The final category of computer-controlled device that we will consider here need a varying voltage to control them. One example of this is a small electric motor that will spin at different speeds, depending on what voltage is applied to it. The opposite of an A/D converter, a digital-to-analogue converter (or D/A for short) changes the digital signals the computer uses into varying voltages. This could, for example, be used to produce sound by connecting it to a loud speaker.

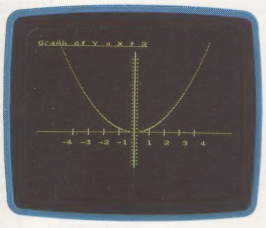
Using microcomputers to control other equipment is just like writing your own software. You have to combine a good idea with some technical knowledge, and a large amount of time. Often the results aren't up to commercial standards, but it is much more fun 'doing it yourself' than buying mass produced products.



ANDY LESLIE

GRAPHIC DESIGN

In this new series, we look at some of the interesting routines that can be created from a few lines of BASIC code. These articles are intended to give you some ideas and start you off on your own programming projects. In this instalment, we see how mathematical graphs can be turned into intriguing designs with simple instructions.



Drawing a graph of a mathematical expression is easy enough. You simply take a range of values for one of the variables in the expression and work out the corresponding values for the other variable. With a simple graph such as $Y = X^2$, we could work out a table like this:

X	-5	-4	-3	-2	-1	0	1	2	3	4	5
Y	25	16	9	4	1	0	1	4	9	16	25

Drawing the points on a piece of squared paper and then joining them up with a smooth curve gives us the familiar necklace-shaped curve. The curve is the graph of $Y=X^2$. Another way of thinking of the curve is that it is the path traced out by a point that is moving along $Y=X^2$. The path is often called a *locus* and by manipulating and combining different loci, we can produce stunning patterns with the minimum of effort.

To do this, let's look at a well-known locus — the circle. The best way to describe a circle with a formula is slightly more complicated than the method used above. Both the X and Y parts of the equation are defined in terms of a third variable or parameter. By varying this parameter through a given set of values, pairs of Xs and Ys can be worked out. The circle is given by the following equation:

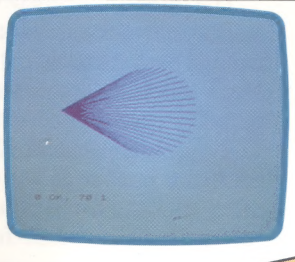
X = R SIN (I)
Y = R COS (I)

If we work out a set of values for X and Y as the angle I is turned a full circle (from 0 to 360°), we get the locus of a circle. The following short program for the Spectrum will create a circle. (See the 'Basic Flavours' box for other BASIC versions.)

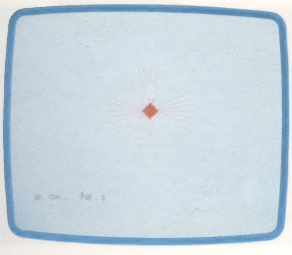
```
10  REM Circle plot
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60 INK 2:  PLOT xc+r*SIN (i),yc+r*COS
(i)
70  NEXT i
```

Note that by changing the STEP size between the points plotted, you can vary the definition of the circle and alter the speed at which it is drawn. Most dialects of BASIC are not fast enough to draw smooth circles from many individual dots at an acceptable speed. To overcome this, it is often preferable to use a number of straight lines to connect the points of the circle. With a large number of straight lines, you can achieve a reasonable compromise between the speed and smoothness of the drawing. You can also use this formula to draw arcs and ellipses. For arcs, you select different values of I. For ellipses, you give R a different value in the X formula than in the Y formula. However, the circle is an adequate basis from which to create some interesting patterns.

Our first pattern is created by drawing a line from a fixed point to every point we plot on the locus. The following programs will position the point in the centre of the circle and to the circle's left respectively.



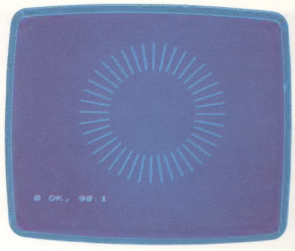
```
10  REM flower
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60 INK 2:  PLOT xc+r*SIN (i),yc+r*COS
(i): DRAW xc-(xc+r*SIN (i)),yc-(yc+r*COS
(i))
70  NEXT i
```



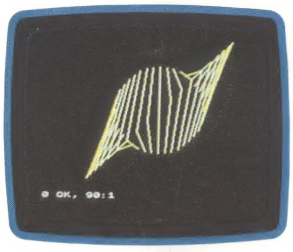
```
10  REM Circle & Fixed point
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60 INK 2:  PLOT xc+r*SIN (i),yc+r*COS
(i): DRAW xc-100-(xc+r*SIN (i)),yc-(yc+r
*COS (i))
70  NEXT i
```

The next step is to use a moving point rather than a fixed one. We will plot two loci simultaneously and draw lines to connect corresponding points on the two paths. The simplest design this will create is two nested circles linked by numerous straight lines.

```
10  REM Nested circles
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60  LET x=xc+(r+30)*SIN (i): LET y=yc+
(r+30)*COS (i)
70  LET p=xc+r*SIN (i): LET q=yc+r*COS
(i)
80  INK 2: PLOT x,y: DRAW p-x,q-y
90  NEXT i
```

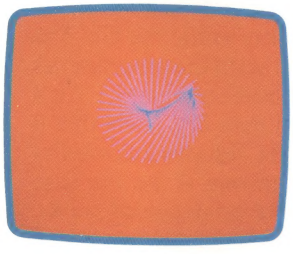


The program we have developed now incorporates enough information to draw hundreds of patterns. All we need to do is to change one or both loci. A simple variation is to swap SIN and COS in one of the formulae. Alternatively, you could use powers of SIN and COS (multiplying them together) to produce different effects.



```
10  REM Twisted circles
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60  LET x=xc+(r+30)*COS (i): LET y=yc+
(r+30)*SIN (i)
70  LET p=xc+r*SIN (i): LET q=yc+r*COS
(i)
80  INK 2: PLOT x,y: DRAW p-x,q-y
90  NEXT i
```

There are many variations on these ideas. A standard mathematics textbook will provide you with the formulae for creating alternative curves. However, you will probably find experimenting with your own programming a more worthwhile pastime. A few simple modifications of the program will even make the computer do the experimenting for you. The final program here cycles indefinitely through a number of randomly-generated patterns — although it will not cover all the possibilities.



```
10  REM Circle & SIN
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60  LET x=xc+SIN (i)*80: LET y=yc+(r+3
0)*SIN (i)
70  LET p=xc+r*SIN (i): LET q=yc+r*COS
(i)
80  INK 2: PLOT x,y: DRAW p-x,q-y
90  NEXT i
```

```
10  REM Circle & SIN*COS
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=50
40  LET s=PI/20
50  FOR i=0  TO 2*PI STEP s
60  LET x=xc+r*SIN (i)*COS (i): LET y=
yc+r*SIN (i)*COS (i)
70  LET p=xc+r*SIN (i): LET q=yc+r*COS
(i)
80  INK 2: PLOT x,y: DRAW p-x,q-y
90  NEXT i
```

```
10  REM Random circle patterns
15  RANDOMIZE
20  LET xm=256: LET ym=176: LET xc=INT
(xm/2): LET yc=INT (ym/2)
30  LET r=60: LET s=PI/20
35  CLS
40  LET c=INT (RND*4)+1: LET d=INT (RN
D*4)+1: LET e=INT (RND*4)+1: LET f=INT (
RND*4)+1
50  FOR i=0  TO 2*PI STEP s
60  LET x=xc+r*SIN (i/c)*COS (i*d): LE
T y=yc+r*SIN (i/e)*COS (i*f)
70  LET p=xc+r*SIN (i): LET q=yc+r*COS
(i)
80  PLOT x,y: DRAW p-x,q-y
90  NEXT i
100  IF INKEY$="" THEN  GO TO 100
110  GO TO 35
```

Basic Flavours

The programs listed here will run on a 16 Kbyte and 48 Kbyte Spectrum. However, converting them and using these ideas on other machines is very simple. Your micro needs high resolution graphics (preferably at least 256×176), a floating point BASIC with the functions SIN and COS, and a command to plot individual points and draw straight lines.

The first adjustments necessary are to XM and YM, the maximum X and Y values that can be plotted on your machine. Depending on the functions you use, you may find that other constants in the program, such as R and S, need to be changed. Next, you must make sure that your micro is in an appropriate graphic mode and select a colour for plotting. Finally, you need a command to draw lines between the co-ordinates given in X and Y and P and Q. On the Spectrum this has to be done with PLOT followed by DRAW. The DRAW command is complicated because on the Spectrum it is always relative to the first point plotted, whereas in this case we need to draw to a particular absolute position. Most micros have an absolute line draw function and this stage is therefore much simpler.

BBC MICRO All the BBC's modes use a grid of 1280×1024 for plotting and you should find that MODE 0 produces spectacular results. Use GCOL to select the plotting colour and MOVE and DRAW to draw the lines.

DRAGON 32/64 The Dragon's PMODE 4 provides a 256×192 grid suitable for these programs. Use SCREEN 1, 0 or SCREEN 1,1 to select either a green or buff background. The LINE command can be used (LINE (X,Y)–(P,Q),PSET) to draw the lines.

COMMODORE 64/VIC-20 These machines have suitable high resolution graphics but unfortunately do not provide appropriate commands. To run these programs, then, you need either to provide your own point and line commands or use a BASIC extension cartridge such as Simon's BASIC.

CAMPUTERS LYNX The Lynx is very suited to this sort of work as it has a full eight colour 256×248 graphics display. Like the Spectrum, this doesn't need a mode command to switch it on. Use INK to select the colour of the design and MOVE and DRAW to draw the lines.

ORIC 1/ATMOS HIRES turns on the Oric's 240×200 graphics screen. The lines can be drawn by using CURSET to move to their start point (X,Y) and then DRAW to draw the line. DRAW on the Oric is relative so the DRAW command has to be of the form DRAW p-x,q-y to work.

Design Ideas

1) Returning to the simple loop to draw a circle, we described how to create arcs and ellipses from the same program. Now see if you can find a way to draw spirals.

2) Try using other functions such as SQR and TAN to generate loci. Be warned that these functions have to be used carefully because they tend to generate awkward numbers. However, you should be able to produce some interesting results.

3) Produce animated versions of the programs. By using arrays to record the last five lines drawn, you should be able to show a group of five lines chasing each other around two loci.

4) How about creating patterns based on three loci? Make two of them very simple (perhaps a circle and a straight line) in order to keep the image uncluttered.

CONTINUING SERIAL

In the last instalment, we introduced the concept of sequential (or serial) files. We looked in detail at how they are constructed and discussed how they are manipulated by the operating system. Here we look at methods for using serial files in your own programs, and show how to overcome some of their inherent limitations.

A sequential file is a solid block of data on a disk or tape, and as such there are limitations as to how the file can be accessed and updated. To retrieve any one item, you must first read through all the preceding data. To update the file, it is usually necessary to make a copy of the file up to the point where changes are needed, then append the alterations to the new file and resume copying the original file immediately after the changes.

It is important to realise that information has to be organised in a suitable way inside the file. The choice of how this is done is up to the programmer and will depend on the application at hand. If it is a file containing English text then it is likely that it will be just a sequence of ASCII codes followed by an end of file marker. However, if the file is to contain a database such as a catalogue of books, then the information needs to be organised appropriately. The common way to do this is to divide the file into *records* and *fields*. Each book has its own record (entry) in the file and within each record are a number of fields, such as the book's title, author, publisher and so on. Within a sequential file, these divisions have to be marked by using special characters placed between items of data.

This is usually done by using a carriage return character (ASCII code number 13) to act as a marker between the fields and records. Since the file will have the same number of fields in each record, it's easy for the program to keep track of where a record ends and a new one begins.

Once a sequential file has been created, you need to be able to access and update it. The basic operations in filing are: retrieving records, adding records, deleting records and amending (editing) records. The diagrams show the various ways of achieving these with sequential files. Because you can only read through a sequential file in order and can't freely change data within it, these operations work by reading through the file, creating a new copy as they go. Any information that is to be changed is then written at the appropriate times into the new file as it is created. Finally, the new file becomes the current file and the old one is either discarded or kept on as a 'back-up' copy.

These simple techniques are the basis of all sequential filing routines. However, they make a major assumption about the capabilities of the operating system — that it can have two different files open at once in order to read from one and write to another simultaneously. This is not possible on all disk systems and is possible only on those cassette-based micros that have two cassette recorders attached. Both the Grundy Newbrain and Commodore PET range feature twin cassette interfaces for exactly this reason. Machines with single cassette drives are limited to files small enough to be read into memory in their entirety and processed there.

These methods of file handling also have an interesting side effect. After any alterations have been made to the file (either additions, deletions or amendments) you have two copies of the file: an old one that was the file before it was updated and a new copy with the changes. It is standard business practice to retain both files so that if something should happen to the new one, there is still a copy that is only one set of changes (or generation) out of date. In fact, most businesses keep three generations of any given file: the new file is called the 'son' file and the preceding file is kept on as the 'father' file. The file that was used as the basis for the father file is known as the 'grandfather' file.

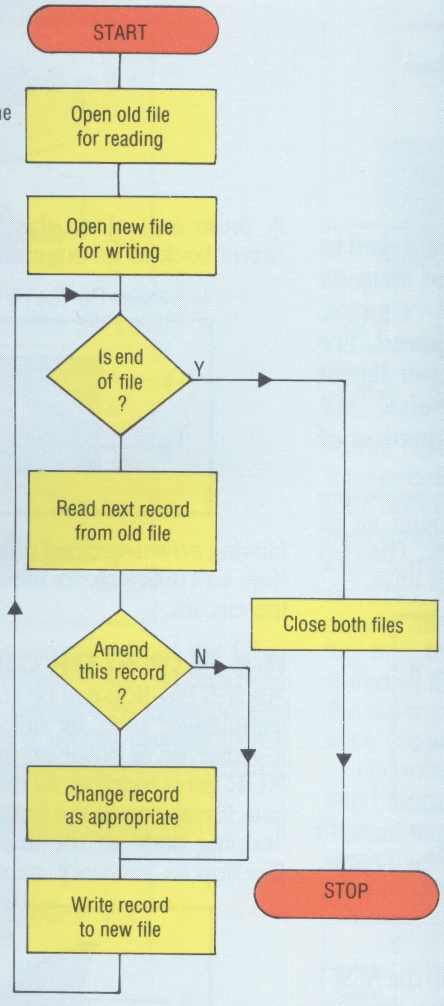
These techniques can work with files that are too big to fit into the computer's memory in their entirety, because only a portion of the file, usually a handful of records, is actually being processed at any one time. With small files, however, much better performance can be gained by reading the whole file into arrays in memory and processing it there. All the file operations can be carried out at high speed in memory before the complete new file is written back to disk or tape.

This approach has one major danger — changes to the file are made permanent only when the information is written back to cassette or disk and, therefore, data could be lost if the program or computer is crashed or switched off while running. If you are using programs that work in this way, you should make sure that you frequently write copies of the file to storage and that a current copy has been made before the program is terminated.

A little experience with sequential file handling will show you that the techniques involved, although cumbersome, are mostly common sense. On many small systems, sequential files are the only file structure provided. When we move on to look at random access files, we'll discover techniques that complement serial files by providing simple and fast access and updating.

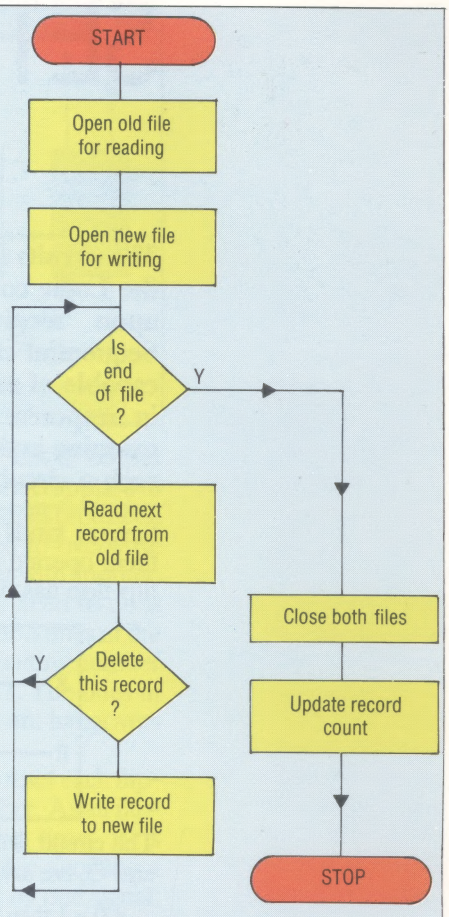
Amending A Record

Amending a record (altering the information within it) is achieved by a combination of deleting and adding. First all preceding records must be copied into a new file. When the record to be updated has been reached and read into RAM, it can be updated by the program. The updated record is written to the new file and all subsequent records from the old file are copied across to the new file. Any number of records can be updated on one pass through the file



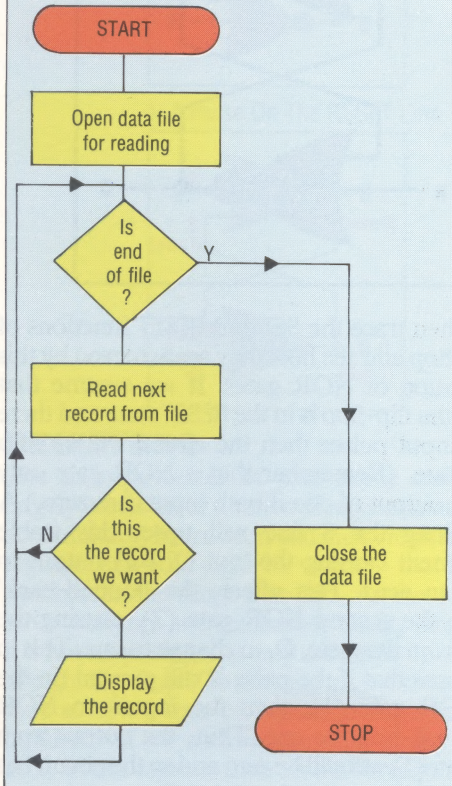
Deleting Records

Records can be deleted from a file by reading and copying through to the record to be removed. This record is then read, but not copied to the new file. Finally, the remainder of the file is read and copied to the new file. A number of records could be removed in a single pass. As when records are added, it is essential that the record count is immediately updated



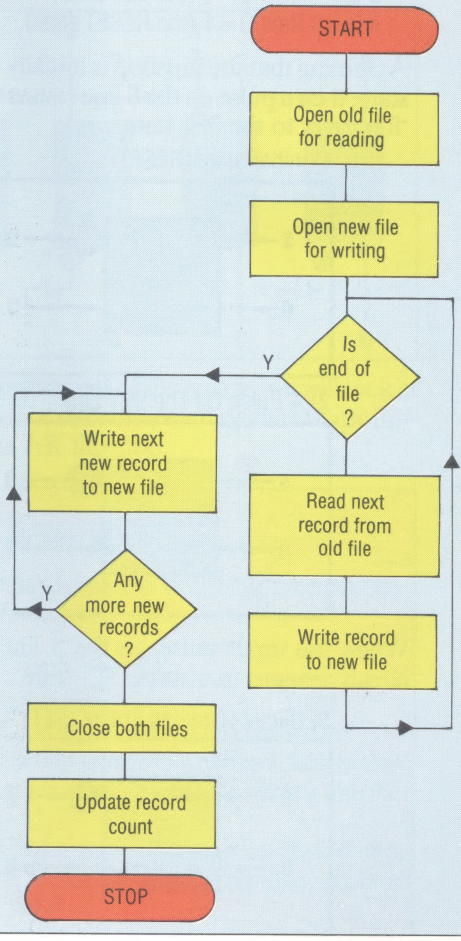
Retrieving Records

Sequential files are not very suited to the sort of application where you fetch old records out of the file. Each time you search for a new one, the whole file has to be read again, which is very time consuming. If you are searching through a list of names for a particular record then your program will just go through a loop, examining each record and carrying on if it doesn't match. If you need a number of records, they can be read one after another but only in the order they occur in the file. For this reason, you often find sequential files are sorted in some order (alphabetically, for example) before they are stored



Adding Records

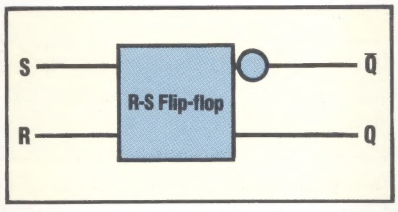
Records can be added to a file in one of two ways. Some BASICs have an APPEND command that lets you add directly onto the end of a file. To add records to files that don't use this command, you have to read through the whole file and produce a copy of it in a new file. Instead of closing the new file, you then write the new records onto the end of it and close both files. In both cases, it is necessary to update the record count. If it's stored with the file, the update routine should make sure that the new value is written to the file immediately so that there is no possibility of losing the information



SET PIECE

The circuits that we have so far considered in the Logic course all produce given outputs upon receiving certain input signals. Sequential circuits, on the other hand, are capable of producing a steady output signal in response to a single input pulse. We examine in detail the design and function of such a circuit — the R-S flip-flop.

Several kinds of flip-flop exist, although all of them operate from the same principles. The R-S flip-flop has two input and two output lines.

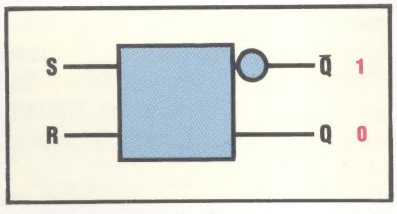


The circuit is designed so that the output lines, Q and $\overline{Q}$, are always opposite to each other. That is:

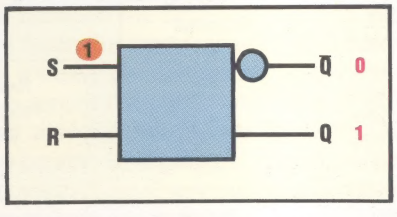
if Q = 1 then $\overline{Q}$ = 0 (the SET state)
if Q = 0 then $\overline{Q}$ = 1 (the RESET state)

Assuming that the flip-flop is initially in the RESET state, then a pulse on the S line causes the circuit to 'flip' over to the SET state.

1) Initial State (RESET)

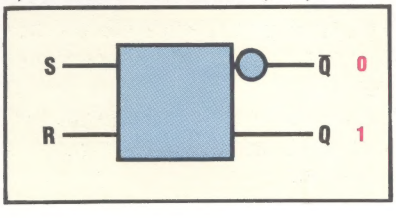


2) A Pulse On The SET Line



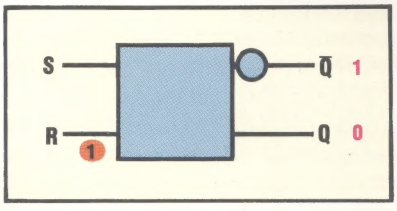
When the input pulse on the S line ceases, the circuit remains in a stable SET state.

3) Circuit Remains Stable (SET)



A pulse sent along the R line flips (flops) the circuit back to its original RESET state.

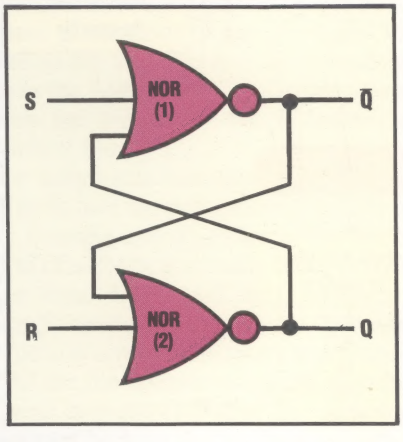
4) A Pulse On The RESET Line



Having now described the function of an R-S flip-flop, let's take a closer look at the logic elements in the circuit.

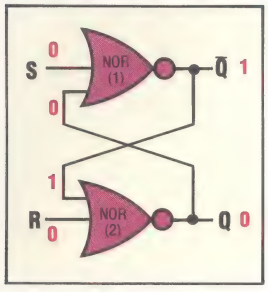
R-S FLIP-FLOP CIRCUIT

An R-S flip-flop can be constructed using several techniques, such as linking two NAND gates together, or, as in our example here, by linking two NOR gates together so that the output from each gate forms one of the inputs to the other. It is this 'looping back' of the logic signals that gives the flip-flop its 'memory' capacity.

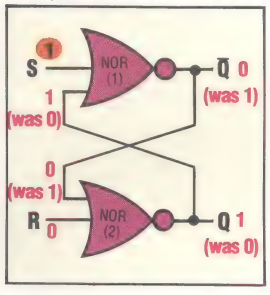


Let us then trace the SET and RESET functions of the flip-flop and see how they are achieved by this combination of NOR gates. If we assume that initially the flip-flop is in the RESET state and there are no input pulses then the circuit will be in a stable state. (Remember that a NOR gate only gives an output of one if both inputs are zero.) A pulse along the S line will upset this stable arrangement causing the 'not Q' ($\overline{Q}$) output to change to zero. This affects the 'looped-back' input to the second NOR gate (2), causing the output from that gate, Q, to change to one. This in turn means that if the pulse is still present on the first NOR gate (1), then the inputs to NOR gate(1) will both be one. Thus, the output from NOR gate(1) will still be zero and so the circuit has reached a stable state, i.e. it is SET.

1) Initial State (RESET)

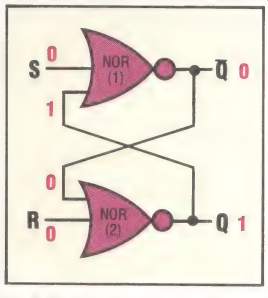


2) A pulse On The SET Line

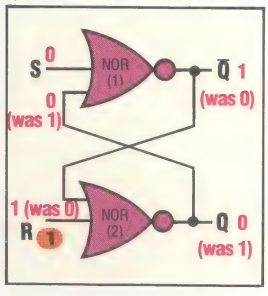


Even when the pulse is removed from the S line the circuit continues in a stable state. When a pulse is sent along the R line, the circuit is again put into an unstable state. After a similar process to that already described, the circuit again settles down to a stable RESET state.

3) Circuit Remains Stable (SET)

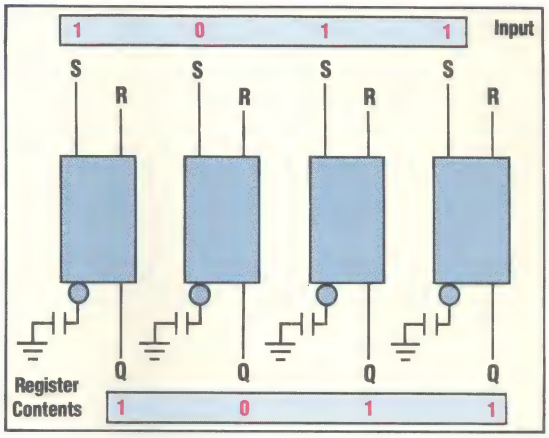


4) A Pulse On The RESET Line



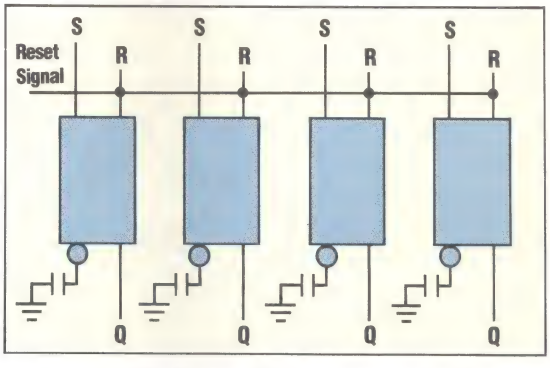
REGISTERS

The microprocessor of your home computer is largely made up of a series of registers, such as the accumulator, instruction and index registers. Most registers can hold eight-bit words — that is, binary numbers in the range 0 to 255. As these registers have to accept and remember binary information it is not surprising that they are made up of a series of eight flip-flops. To simplify matters, we will here look at how a four-bit register accepts and stores numbers. If we wish to store the binary number 1011 in the register then all that is required is for the binary pattern to be fed to the S lines.



Notice that in this arrangement the 'not Q' output is unused. As the binary input pattern is applied to the S lines of the flip-flops, so the Q lines produce a corresponding output. If we wished to overwrite the first number stored in the register with another, say 0110, we may think that all we need to do is present this new binary pattern to the S lines of the flip-flop. In fact, if we did this the resulting number stored in the register would be 1111. The ones in the outer positions of this number are hangovers from the previous number.

The solution to this problem is to reset each flip-flop before storing the second number. As all flip-flops need resetting at the same time, it is convenient to connect them together allowing a reset of the register to be triggered by one signal.



LIZ DIXON

In the next instalment of the course, we will look at other sequential circuits, including the D-type flip-flop and the J-K flip-flop.

Exercise 7

1) Why is a flip-flop also known as a 'bistable'?

2) Initially, when a computer is switched on, a flip-flop is in the following state:

$Q = 0, \overline{Q} = 0, S = 0, R = 0$

a) Is this a stable state?
b) If not, what state will the flip-flop change to?
c) Can the flip-flop change to a different state to that given in your last answer? (Hint: try starting with the other gate.)
d) What process is necessary, when you switch on a computer, to ensure that all the registers are in a predictable state?

RIFLE RANGE

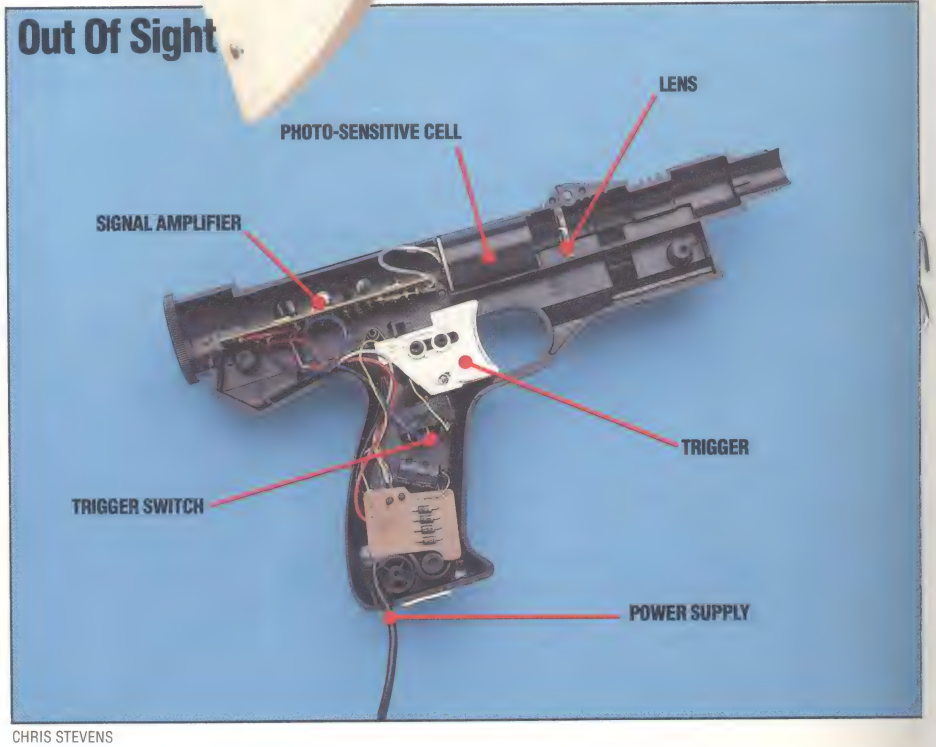
The Stack Light Rifle (SLR) is designed to bring added realism to 'shoot-em-up' games on home computers. Combining the appearance of a gun with a camera-style optical system, the SLR is hardly a precision instrument, but its use of light pen technology allows the user to dispense with joystick or keyboard control.

The main component of the Stack Light Rifle System is the electronic target pistol that is connected to the computer by a generous length of lead. At the computer end, depending on the version, there is a connector for the appropriate socket or edge connector. On the ZX Spectrum version the connector contains two chips and a couple of simple components to interface the main electronics inside the gun to the computer. To make the pistol more accurate—and to turn it into a rifle— it is supplied with a shoulder stock that clips and secures to the rear of the pistol, a barrel and a make-believe telescopic sight.

The electronics inside the pistol consist of a light detector or photo-diode and a small amplifier and buffer. Light coming down the barrel is focused by a small plastic lens onto the photo-diode, and the device is sensitive enough to detect the changes in intensity of the picture. Once boosted by the amplifier, the signal is clipped to provide a digital pulse rather than an analogue waveform and is then fed to the computer via the switch. The screen position that is being scanned at that moment is the position the rifle is pointing at. As the computer receives the pulse from the Light Rifle it compares the value of its scan registers with the screen position of the target and, if a match is found, the played has scored a direct hit.

Variants of the Light Rifle are currently available for the ZX Spectrum, Commodore Vic-20 and Commodore 64 and all perform the same function. Stack provides three games on cassette with the Light Rifle but that's about the limit of the support provided. Although various independent software houses produce games that would appear to be eminently suited to this type of user control, very few have actually produced or converted programs to work with it; Micromania is an exception. Possibly even more damaging to potential sales of the Light Rifle is the fact that Stack doesn't provide any driver routines to allow users to write their own programs. This omission, together with the lack of any technical details on how it works, means that the Light Rifle is not a good alternative to a joystick.

The Light Rifle is based on the same principle of



CHRIS STEVENS

operation as a light pen, but is much bigger and is designed to be held up to about three metres (10 feet) from the television set rather than in contact with the screen. To help filter out any ambient light, the Light Rifle is provided with both a long dark tube (the barrel) and a lens. These combine to provide a reasonable — if not perfect — degree of accuracy, and allow the user to 'shoot-em-up' from the comfort of an armchair. The games that are supplied are rather poor examples of what should be possible; both the use of graphics and the 'playability' are hardly outstanding.

One of the major problems in programming light pens, or even giant versions such as the Light Rifle, is that the program needs to be very efficiently written. In all the examples supplied by Stack, the games come to a halt when the trigger is pulled. This is because the requirement of continuously scanning the screen, as is usually done for a light pen, would slow the games down too much. So when the trigger is pulled on the Light Rifle, the software must freeze the action and establish whether the target on the screen is aligned with the position of the gun. Once the software has determined whether or not the player has hit the target, the game can continue. In theory, when the trigger has been pulled, the amount of code necessary to establish the screen position of the next scan detected by the gun should be very small indeed, but observing the software in action indicates that this isn't always the case.

On a computer such as the BBC, for which there is as yet no version of the Light Rifle, the provision of a light pen facility within the video chip would make the task of the software much simpler. The Commodore 64 offers such a system, but the ZX Spectrum, on which the Light Rifle was tested, lacks the facility and the deficiency shows up in the time taken to calculate the position of the rifle when the trigger is pulled.

Shot In The Dark

The photo-cell in the Light Rifle detects the moving dot of the raster scan as it refreshes the television picture. The software continually monitors the raster counter register in the computer, so that it always knows the position of the raster on the screen; when the light rifle signals that it has detected the raster, the software can convert the raster counter value into the X,Y screen co-ordinates of the Rifle's point of aim

Open Season

High Noon is the best of the three Stack demonstration programs. Animation is good, and the gunfighter shoots back credibly. Grouse Shoot and Shooting Gallery both provide a single moving target that must be hit before it goes out of bounds. Animation is jerky, and pulling the trigger causes a noticeable halt in screen action

IAN McKINNELL

DEFINING TERMS

The Commodore graphics set is extensive, but it is often necessary to create some special characters, or even to redefine the entire character set. In this instalment of our graphics series we introduce the techniques of user-defined graphics on the Commodore 64 and continue to develop the Subhunter game.

The process of creating your own characters on the Commodore 64 is not straightforward: there are no special-purpose commands in Commodore BASIC, so the whole operation has to be carried out using PEEK and POKE to access and change the contents of memory.

The Commodore 64 character set consists of a block of ROM starting at memory location 53248. Each character appears on the screen as a pattern of dots in an eight by eight dot matrix: describing this pattern of 64 dots requires 64 bits, or eight bytes. The eight bytes from location 53248 to 53255 describe the '@' character, the first character in the set; it has a screen code of 0, which means that if you POKE the value zero into one of the bytes of video RAM, this character will appear on the screen. The next eight bytes, from 53256 to 53263 describe 'A' (screen code 1), and so on.

We cannot change these dot matrix definitions in ROM, so we must copy some or all of them into RAM and make the changes there. We can then make the Commodore use our RAM character set for writing on the screen, rather than using its own definitions in ROM.

The ROM character set shares its address space in memory with input/output devices such as cassette players and disk drives. Normally, the 6510A CPU treats this memory space as an input/output area, but it can be programmed to regard it as the character set location. This may seem strange, but the CPU doesn't normally do the work of accessing character definitions from ROM and sending them to the screen. That task is delegated to a subsidiary chip under CPU control. The contents of location 1 determine the status of I/O operations, and bit 2 of this location acts as a switch on the way in which the CPU regards the character set ROM. If this bit is set to zero, then the CPU finds the I/O devices occupying the space. The other bits of location 1 have similar special functions in controlling the system, so we must be careful not to alter any of them while changing bit 2's value. This is best achieved by using the logical operators AND and OR.

Suppose that the contents of location 1 are:

Bit	7	6	5	4	3	2	1	0
	0	1	1	0	1	1	1	1

We wish to change bit 2 to zero. One way to do this would be to calculate the decimal value of 01101011, and POKE it into location 1, but this works only if we know that the previous contents of location 1 were 01101111. A better way to adjust bit 2 is to use AND and PEEK. The following command PEEKs location 1, thus establishing its original contents, ANDs them with 251 (11111011 binary), and POKEs the result back into location 1:

```
POKE 1,PEEK(1) AND 251
```

The effect of this command can be illustrated here:

Bit	7	6	5	4	3	2	1	0	
	0	1	1	0	1	1	1	1	= Initial contents
AND	1	1	1	1	1	0	1	1	= 251 binary
	0	1	1	0	1	0	1	1	= Result of ANDing each pair of bits

No matter what the original value of bit 2, ANDing it with zero will always produce a zero result; ANDing all the other bits of the location with one simply produces a copy of their original value. The binary number 1111101 (251 decimal) is called a *mask* or overlay, and here we are using it as an 'AND-mask'.

To set bit 2 to one without affecting any of the other bits, we use the following command:

```
POKE 1,PEEK(1) OR 4
```

Bit	7	6	5	4	3	2	1	0	
	0	1	1	0	1	0	1	1	= Initial contents
OR	0	0	0	0	0	1	0	0	= 4 binary
	0	1	1	0	1	1	1	1	= Result of ORing each pair of bits

This ensures that BASIC will not overwrite our character set. When the copy is complete, the CPU can be reset to address the I/O devices, and the interrupt time re-started.

The final piece of the jigsaw is forcing the screen-handling chip to use our character set, rather than the system set in ROM. Bits 0 to 3 of location 53272 point to the start address of the character set, and the following table shows how the Commodore 64 interprets the values of these bits as pointing to particular addresses:

decimal value of bits 0-3	bits 3,2,1,0	location pointed to
0	0000	0
2	0010	2048
4	0100	4096
6	0110	6144
8	1000	8192
10	1010	10240
12	1110	12288
14	1110	14336

The value of bit 0 in this register is unimportant, while bits 4 to 7 control other functions, and must be left unchanged. We use 11110000 (240 decimal) as an AND-mask for this purpose, and 00001110 (14 decimal) as an OR-mask to make the register point to 14336 — the start location of our character set:

```
POKE 53272,(PEEK(53272) AND 240) OR 14
```

Now, using a FOR..NEXT loop we can perform the actual copying.

While a program is copying the ROM character set into RAM, the CPU cannot deal with I/O device interrupts. The keyboard, for example, interrupts the CPU every sixtieth of a second, causing it to scan the keyboard for a keypress. These interrupts are triggered by the system timer. If the CPU were interrupted by an I/O device while the character set was occupying the I/O ROM space (as is the case during copying), then the system would probably crash, and only turning the power off and on would reset the machine. Fortunately, we can disable the interrupt mechanism by setting bit 0 of location 56334 to zero; the other bits of this location must be left unchanged, so the following logical POKE command should be used:

```
POKE 56334, PEEK(56334) AND 254
```

Once the interrupts are disabled, and the CPU is forced to look at the character set in ROM, then copying can begin.

We wish to copy, say, the 64 characters from '@' to '?' — screen codes 0 to 63 — so we must copy the 512 locations (8×64=512) starting at 53248 into a suitable block of RAM. Various areas can be used, and our choice here is a block starting at location 14336. This would normally be in the BASIC program area but we can protect it by lowering the top-of-memory pointer in location 56, thus:

```
POKE 56, 32
```

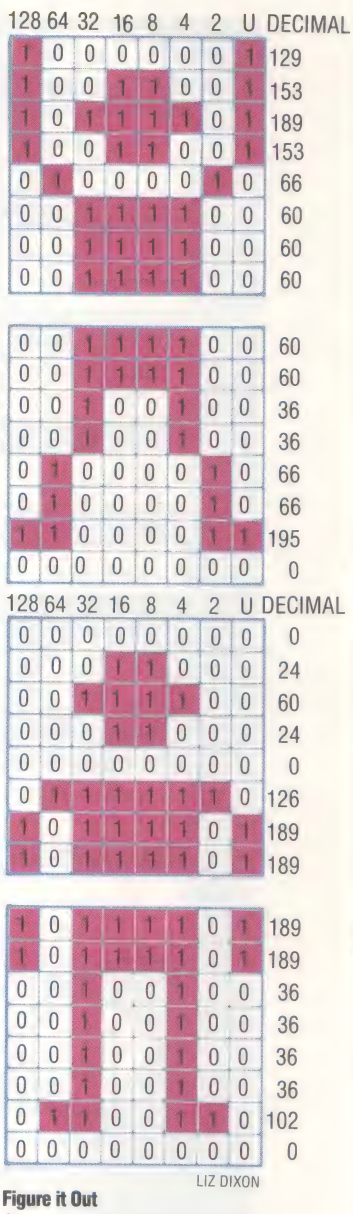
DESIGNING NEW CHARACTERS

Each character in the Commodore set is designed on an eight by eight dot matrix. Each row of the matrix is interpreted as a binary number (dots that are illuminated count as one, dots that do not show up on the screen count as zero) and so requires a byte of storage, and the entire eight-row character requires eight consecutive locations in memory. The starting location of the bytes describing any character can be calculated from the start address of the entire block, and the screen code of the particular character, thus:

Start of character = 14336+8×(screen code)

Once the locations of the bytes defining a character are known, we can POKE new values into those bytes, thus changing the dot patterns appearing on the screen when that character is printed. As long as the one character set is enabled, pressing the appropriate key will cause the new character to appear on the screen. In the demonstration program, we redefine the characters [,],£, and ↑ (codes 27-30) as parts of a figure, and achieve animation by printing and overprinting different versions of the figure.

```
130 :
140 REM**** COPY ROM CHAR SET ****
150 POKE56,32
   :REM LOWER TOP OF MEMORY
160 POKE56334,PEEK(56334)AND254
   :REM TURN OFF INTERRUPT TIMER
165 POKE1,PEEK(1)AND251
   :REM FLIP TO CHAR ROM
170 FOR I=0 TO 511
   :REM COPY
180 POKE14336+I,PEEK(53248+I)
   :REM 64
190 NEXTI
   :REM CHARACTERS
200 POKE1,PEEK(1)OR4
   :REM FLIP BACK TO I/O
210 POKE56334,PEEK(56334)OR1
   :REM TURN ON INTERRUPT TIMER
220 POKE 53272,(PEEK(53272)AND240)
OR14:REM SET CHAR POINTER
230 REM****  COPY COMPLETE  ****
240 :
250 :
300 REM**** ATHLETIC ARTHUR ****
310 FOR I=14552TO14552+31
   :REM READ
320 READ A:POKEI,A:NEXT I
   :REM CHAR DATA
330 PRINTCHR$(147)
   :REM CLEAR SCREEN
340 POKE55338,14:POKE55378,14
   :REM COLOUR CHRCTR. LIGHT BLUE
350 POKE1066,27:POKE1106,28
   :REM ARMS UP
360 FORI=1TO500:NEXT I
   :REM DELAY LOOP
370 POKE1066,29:POKE1106,30
   :REM ARMS DOWN
380 FORI=1TO500:NEXT I
   :REM DELAY LOOP
390 GOTO350
480 :
490 :
500 REM**** ARMS UP DATA ****
510 DATA129,153,189,153,66,60,60,60
520 DATA60,60,36,36,66,66,195,0
530 REM**** ARMS DOWN DATA ****
540 DATA0,24,60,24,0,126,189,189
550 DATA189,189,36,36,36,36,102,0
```

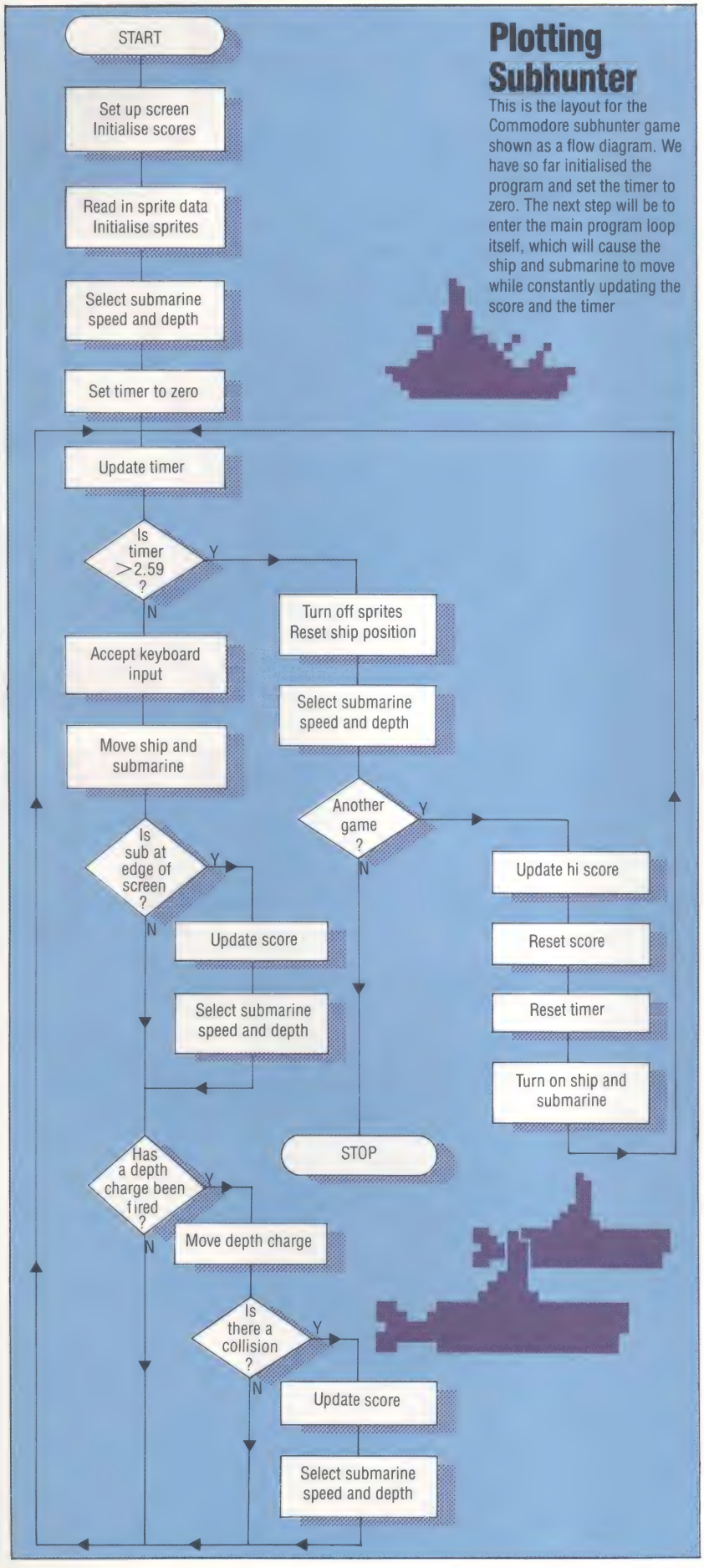


LIZ DIXON

Figure it Out
Characters are built in an eight by eight dot matrix, described in eight consecutive bytes. Each row of the character is interpreted as a single-byte binary number, dots representing ones and spaces representing zeros. For use in Commodore 64 programs, these binary numbers must then be converted to decimal

Plotting Subhunter

This is the layout for the Commodore subhunter game shown as a flow diagram. We have so far initialised the program and set the timer to zero. The next step will be to enter the main program loop itself, which will cause the ship and submarine to move while constantly updating the score and the timer



Subhunter Program

The Commodore 64 has its own internal clock that can be used to time BASIC programs. The clock has six digits, rather like a digital watch, representing hours (00-23), minutes (00-59), and seconds (00-59). The clock can be accessed from BASIC through the string variable TI$. The value of TI$ gives the time that has elapsed since the computer was turned on, but it can also be reset at any time. The following short program demonstrates how the timer works.

```
10 REM ....TIMER....
20 PRINT CHR$(147)      : REM CLEAR SCREEN
30 TI$ = "000000"       : REM SET TIMER TO
                          ZERO
40 PRINTCHR$(145);TI$ : REM PRINT CURRENT
                          VALUE OF TIMER
50                      : REM CHR$(145)=
                          CURSOR UP
60 GOTO40
```

The program runs in a continuous loop, printing the timer to the screen until you press the Run/Stop key.

The Subhunter game we are writing requires a clock to be displayed on the screen and to end the game when three minutes have elapsed. The game clock, therefore, requires only the minutes and seconds parts of TI$. By using the string functions we can break TI$ down as follows:

RIGHT$ (TI$,2)

TI$=HH(MM)(SS)

MID$ (TI$,3,2)

The two seconds digits can be stripped off by RIGHT$(TI$,2), and the minutes digits can be isolated by MID$(TI$,3,2).

The main program loop of our game starts at line 200 and ends at 390. Load up the subroutine already typed in from the last section and add these lines:

```
140 TI$="000000"
150
160
200 REM .... MAIN LOOP ....
205
210 REM .. TIMER ..
220 PRINTCHR$(19); TAB(14)CHR$(5)"TIME
    ";MID$(TI$,3,2);":";RIGHT$(TI$,2)
225 IFVAL(TI$)>=259THEN400:REM END GAME

390 GOTO200:REM RESTART MAIN LOOP
400 END
```

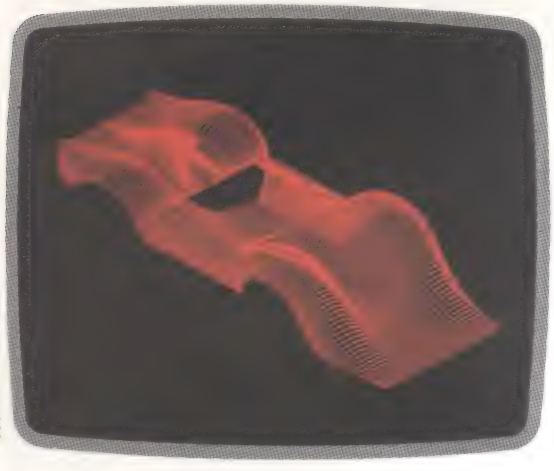
Line 140 re-sets the clock at the start of the program. Line 220 PRINTs the current value of the clock in minutes and seconds, separated by a colon. TAB(14) causes 14 spaces to be left before PRINTing and positions the clock in the middle of the screen. CHR$(5) will colour the characters white. Line 225 converts TI$ to a numeric quantity so that its value can be tested. If playing time has exceeded two minutes and 59 seconds, then the game is at an end.

C

CAD/CAM

Computer aided design and *computer aided manufacturing* are both terms that are more commonly referred to by their initials, and are frequently considered together as a single concept. In this context, the word 'design' generally refers to engineering or functional design, rather than aesthetic design — though computers are now making inroads into that field as well.

There are three main advantages of using computers to aid in design. The first advantage concerns the whole area of graphical visualisation of the product being developed. Most CAD systems feature both a high resolution screen display and a full colour plotter for hard copy



COURTESY OF INTERGRAPH

print-outs. The software used by such a system is really the graphical equivalent of a word processor package, permitting parts of an image to be amended, deleted, moved around or called up from a disk library of standard components. The more advanced models, which 'think' in three dimensions, allow a design to be viewed from any direction or angle.

Secondly, a computer can perform all the tedious and time-consuming calculations required in the background, leaving the designer free to concentrate on the higher level decisions. Such calculations might include checking for clearance and stress, or evaluating the functions that define a curve used in the projected design.

The third benefit involves taking this a stage further — using the computer to find the optimum design within given constraints. This may be optimising for weight, size, strength or cost. In many cases there is no formula for the optimal solution, but the computer can perform trial and error calculations thousands of times faster than an engineer.

Computer aided manufacturing is a very broad term, arguably encompassing the whole field of robotics. There is now a general trend in industry away from fixed production lines towards flexible manufacturing systems (FMS), which can be easily re-programmed. This allows for the production of goods to be more closely related to sales. Another important aspect of CAM is CNC — *computer numerically controlled* machine tools. A single CNC machine can replace dozens of fixed-position drills on a production line.

CAI/CAL

The use of computers in education is a controversial subject, even among those who support their introduction into schools. The debate centres on whether computers are best employed as aids to either teaching or learning. This distinction is made clearer by considering the differences between CAI and CAL.

Computer aided instruction (CAI) can be defined as applying the computer to traditional methods of instruction. The computer is regarded as a combination of electronic textbook and electronic tutor. Typically, the student will work through a program on his or her own. The computer breaks down each lesson into modules, perhaps using graphics, animation or sound to make the material more interesting than it would be in a textbook or on a blackboard. At the end of each module, the computer tests the student using multiple-choice questions to find out how well the material has been understood. It allows each child to progress at his or her own pace, can backtrack when appropriate, and can keep a record of the child's progress for the teacher to study.

Proponents of *computer aided learning* (CAL), however, argue that for the first time the microcomputer allows us to place the emphasis on learning rather than on being taught, which may have been the only practical approach in the past owing to limited resources. CAL applications use the processing power of the microcomputer to create an environment in which the child can explore and learn — in much the same way as very young children learn about the world from playing with sand and water.

There can be little doubt that computer aided learning results in a better understanding on the part of the child, but it must be remembered that CAL applications require a great deal of imagination and considerably more sophisticated programming than their CAI counterparts.

CALL

`CALL` is a programming keyword that instructs the computer to invoke another routine. Your computer may not feature this word in its dictionary, but it will certainly have an equivalent command. `GOSUB` is a form of `CALL`, as are the commands: `PROC`, `SYS` and `USR`. Another common use of this keyword is to mean 'go into machine code from this point'. Effectively, the programmer is calling up another subroutine, but this one is written in machine code.

The term originates from the early days of programming, when most programs were assembled by stringing together subroutines from a large library of functions called a 'macro library'. When the `CALL` instruction was encountered, the computer literally had to 'call up' the lines of the appropriate subroutine from a disk file.

GENERAL ROUTINE

Making machine code programs relocatable, so that their execution is independent of their locations, requires the use of symbols and labels rather than absolute addresses and values. We study some more assembler directives and their role in program structure, and take a first look at Assembly language subroutine calls.

Because Assembly language is essentially a simple programming language composed of the 'primitive' commands that the CPU can manage, you will find yourself constantly writing and re-writing fragments of program to do the same essential tasks that you take for granted as part of the instruction set of a high-level language — input/output handling, for example, or two-byte arithmetic routines. The sensible thing to do is to establish a library — on tape, disk or paper — of the most commonly used routines, and merge these into new programs as the need arises.

There are two major problems associated with this, however. The first is the difficulty of writing important, and often lengthy, routines in a sufficiently general way that they can be inserted in different programs without adjustment or re-writing. The second problem is in writing useful routines that are not rooted in one set of memory locations, so that they can be relocated in memory through a new assembly with a different ORG address, and perform exactly the same function there as in their original locations.

Both problems are aspects of the generality/portability problem familiar to BASIC programmers, and are solved in much the same way — by using variables to pass values from program to subroutine; by using local variables in subroutines to make them independent of the larger program context; and by avoiding the use of absolute quantities (both numerical or string constants) and program line numbers.

In Assembly language programming we have become used to the idea of memory locations as the equivalent of BASIC variables — programs operate on the contents of the locations, whatever those contents might be, in the same way that a BASIC program operates on the contents of its variables. Unfortunately, we have tended to refer to memory locations by their absolute addresses, a convenient habit at first, but one that must now be renounced in the name of generality. The answer is to use symbols instead of absolute addresses and values, and to use the range of symbolic forms offered by assembler pseudo-opcodes as the equivalents of both variables and program line numbers. We have seen examples of both uses already. Consider this program, for example:

6502			Z80		
DATA1	EQU	$12	DATA1	EQU	$12
DATA2	EQU	$79	DATA2	EQU	$79
	LDA	DATA1	LD	A,(DATA1)	
LOOP	ADC	DATA2	ADC	A,(DATA2)	
	BNE	LOOP	JR	NZ,LOOP	
	RTS		RET		

Here we have two kinds of symbol, two values and a label, all used as the operands of the Assembly language instructions. Because of this, the program fragment is both general and able to be relocated. The only absolute quantities are the values of DATA1 and DATA2, and they can be initialised in the surrounding program, rather than at the start of the routine itself.

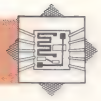
There are other pseudo-ops that we have not yet discussed. In particular, DB, DW and DS (though, like ORG and EQU, they may differ from one assembler program to another). These three directives, which stand for 'Define Byte', 'Define Word', and 'Define Storage', enable us to initialise and allocate memory locations, as in this example:

			ORG	$D3A0
D3A0	5F	LABL1	DB	$5F
D3A1	CE98	LABL2	DW	$98CE
D3B3		LABL3	DS	$10
D3B3		DATA1	EQU	LABL3

SYMBOL TABLE:
LABL1 = D3A0: LABL2 = D3A1: LABL3 = D3A3
DATA1 = D3A3
ASSEMBLY COMPLETE — NO ERRORS

In this full Assembly listing (the output of an assembler program) we see at the bottom for the first time a symbol table, consisting of the symbols defined in the program and the values they represent. There are several important things to notice in this fragment. First of all, in the line that begins LABL1, the DB pseudo-op is used. We can see from the listing that the ORG directive has given the address $D3A0 to LABL1, and the symbol table confirms this. The effect of DB here is to place the value $5F in the byte addressed by LABL1 — so memory location $D3A0 is initialised with the value $5F, as we can see in the machine code column of the listing.

Secondly, LABL2 represents the address $D3A1. However, DW has the effect of initialising a 'word' (two consecutive bytes) of storage, so the value $98CE is stored in locations $D3A1 and $D3A2 in lo-hi form — this can be seen clearly in the machine



code column. Because DW automatically converts its operands into lo-hi form, it is most often used to initialise 'pointer' locations with addresses. LABL2, or location $D3A1, might be such an address — it points to location $98CE.

The third thing to consider is that the instruction DS $10 has the effect of adding $10 to the program counter. This is clearer in the symbol table than in the actual listing — LABL3 represents the location $D3A3 (the location following the previous instruction), though it appears from the listing that its value is $D3B3. This is actually the location address of the next instruction after the DS instruction, so DS $10 has reserved a block of 16 bytes (from $D3A3 to $D3B2 inclusive) between one instruction and the next. This is a process rather like putting long REM lines into a BASIC program to create unused space in the program text area that can then be POKEd and PEEKed as a machine code program area (see page 137).

Finally, the last instruction uses EQU to set one symbol equal to the value of another, so that DATA1 has the value $D3A3 (the value of LABL3). This is another source of possible confusion. LABL3 is the symbolic representation of the location address $D3A3, so DATA1 EQU LABL3 means 'the symbol DATA1 is to have the same meaning and value as the symbol LABL3'. The fact that the DB instruction has made the contents of $D3A3 equal to $5F has no significance for the meaning of the symbols LABL3 and DATA1. Keeping the distinction between a location and its contents clear in your mind is one of the most testing difficulties in the early stages of learning Assembly language programming. You may have had the same problem with BASIC program variables and their contents.

At first glance, the DB directive seems to duplicate EQU, but this is not the case. LABL1 means 'the location $D3A0', and DB $5F has initialised that byte with the value $5F, but, although the value of LABL1 is now fixed, the contents of the location it symbolises can be changed at any time (by storing the accumulator contents there later in the program, for example). Similarly, DATA1 is now a symbol whose value is fixed by the EQU instruction; its value cannot be changed by the program's execution. And again, LABL3 points to the start of a 16-byte data area, the contents of which can be changed in the program, but LABL3 is itself unchangeable.

This introduces, but does not exhaust, the possibilities of the new pseudo-ops. Consider this new version of the previous fragment:

```
                        ORG   $D3A0
D3A0    4D4553  LABL1   DB    'MESSAGE 1'
D3A9    CE98    LABL2   DW    $98CE
D3BB            LABL3   DS    $10
D3BB            DATA1   EQU   LABL3

SYMBOL TABLE:
LABL1 = D3A0: LABL2 = D3A9: LABL3 = D3AB
DATA1 = D3AB
ASSEMBLY COMPLETE — NO ERRORS
```

The DB instruction has a string, 'MESSAGE 1', as its operand, and the assembler has initialised the locations from $D3A0 to $D3A8 with the ASCII values of the characters within the single quotes. This can be inferred from inspection of the location address column in the listing, and is partly confirmed by the machine code column — the contents of the three bytes from $D3A0 to $D3A2 are shown to be $4D, $45, and $53, which are the hex ASCII codes for 'M', 'E', and 'S'.

This is a significant facility, not only because it relieves the programmer of the task of translating messages and character data into lists of ASCII codes, but also because it makes the listing much easier to read, and hints at the possibility of actually getting some screen output from our Assembly language programs. The latter is particularly significant because so far we have been restricted to storing results in memory and inspecting them using the Monitor program (see page 118). Naturally, we will be exploring screen-handling in the course, but there are still aspects of Assembly language that we need to investigate before going onto that topic. If, however, you think about our habit of storing results in memory, and if you understand already that memory-mapped screen displays are, in effect, only areas of memory, then you may be able to see a way of addressing the screen from a program.

The most important aspect of this new DB facility is that it confers on LABL1 the status of a

Exercises

1) The first program fragment in the main text uses the DS pseudo-op to reserve $10 bytes of memory starting from the address represented by the label LABL1. Write an Assembly language program that will store the numbers $0F to $00 in descending order in this block, one number per byte. This should be done using a loop, and indexed addressing techniques, for which you will need to use the DEX (decrement the X register) or DEC (IX+0) (decrement IX) instructions. The loop should continue as long as decrementing the index register does not cause the zero flag to be set, so use the BNE or JR NZ branch instructions.

2) Using the techniques of the previous exercise, write a program to copy the message stored at LABL1 by the DB pseudo-op (see the second program fragment in the main text) to a block of memory starting at the address stored at LABL2 by the DW pseudo-op. The address $98CE may not be suitable for your computer, so change the initialisation, but the program should work for any address, and for any length of message. To implement this, your program must use either the number of characters in the message as a loop counter, or it must be able to recognise the end of the message — you might put an asterisk, for example, as the last character of any message.

BASIC string variable: When we write in BASIC:

```
200 LET A$="MESSAGE 1"
```

then we are actually creating a pointer to the start of a table of bytes containing the ASCII codes for 'M', 'E', 'S', and so on. Whenever the BASIC interpreter encounters a reference to A$, it looks in its own symbol table to find the location at which it points — that is, the starting location of the contents of A$. Similarly, in our Assembly language program we can treat LABL1 as the equivalent of A$, given that we have already written a program fragment that allows us to manipulate a table using indexed addressing.

The pseudo-ops, then, allow us to remove absolute addresses and values from our programs, and replace them with symbols. This has the effect of diminishing the problems of portability and relocatability. What we need now is to be able to access these portable, relocatable modules from the main program. In other words, we need a machine code equivalent of BASIC's GOSUB command.

There is such an instruction, of course: JSR and CALL in 6502 and Z80 respectively. Both require an absolute address (which can be a label) as operand, and both have the effect of replacing the contents of the program counter with the address that forms their operand. The next instruction to be executed, therefore, will be the first instruction of the subroutine so addressed. Execution continues from that instruction until the RETURN instruction — RTS and RET respectively — is encountered. This command has the effect of replacing the current contents of the program counter with its contents immediately prior to the JSR or CALL instruction was executed. The next instruction to be executed, therefore, is the instruction immediately following the JSR or CALL. This is exactly the mechanism used by the BASIC interpreter in executing and returning from GOSUBs. It's easily understood as such, but it raises the question of how the old contents of the program counter are restored when the RETURN instruction is executed. The simple answer is that the JSR and CALL instructions first 'push' the program counter contents onto the stack (see illustration on page 136) before replacing them with the subroutine address; and the RTS and RET instruction 'pop' or 'pull' that address from the stack back into the program counter. The questions of what the stack is, how you push or pop it, and why you'd want to do so, are the subject of the next instalment of the course.

Instruction Set

BEQ — BRANCH ON ZERO — 6502

Relative F0 (2 bytes)

The contents of the program counter are offset by the value of the byte following the op-code.

EFFECT ON PSR

	S	V		B	D	I	Z	C
MSB								LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
8F00	F0 16	BEQ $16

	BEFORE		AFTER
Program Counter	02	lo	18
	8F	hi	8F

F0	$8F00
16	
	Program Memory

JR Z — JUMP RELATIVE ON ZERO — Z80

Relative 28 (2 bytes)

The contents of the program counter are offset by the value of the byte following the op-code.

EFFECT ON PSR

	S	Z		H		V	N	C
MSB								LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
8F00	28 16	JR Z,$16

	BEFORE		AFTER
Program Counter	02	lo	18
	8F	hi	8F

28	$8F00
16	
	Program Memory

INX — INCREMENT X REGISTER — 6502

Implicit E8 (1 byte)

The contents of register X are increased by one.

EFFECT ON PSR

	S	V		B	D	I	Z	C
MSB	X						X	LSB

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
F391	E8	INX

	BEFORE	AFTER
PSR	????????	0??????1
X	FF	00

E8	$F391
	Program Memory

INC IX — INCREMENT IX — Z80

Implicit DD23 (2 bytes)

The contents of IX are increased by one.

EFFECT ON PSR

	S	Z		H		V	N	C
MSB								LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
F391	DD23	INC IX

	BEFORE		AFTER
Program Counter	FF	lo	00
	E7	hi	E8

DD	$F391
23	
	Program Memory

LIZ DIXON

POPULAR CHOICE

Since its introduction, the Control Program for Microprocessors (CP/M) has become the industry standard in operating systems. The phenomenal success of CP/M has changed the life of its designer, Gary Kildall, who left the teaching profession to found the company known as Digital Research.

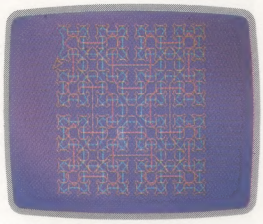
Gary Kildall, a member of the Intel team that developed the 8080 microprocessor, created his first version of the CP/M system in 1974 to support a compiler for PL/M, the first high-level language produced by Intel. In 1975, he added an editor (ED), assembler (ASM), and debugger (DDT). He offered the new operating system to Intel, who turned it down—which was probably Kildall's luckiest break. Partnered by Dorothy McEwan, he started to publish hobbyist magazines and to sell CP/M privately. Kildall's CP/M quickly outsold the hobbyist magazines.

Whether by design or sheer good luck, Kildall had hit upon a system that greatly diminished the major problem of the microcomputer in its early years—compatibility. The three most significant consumer computers of the late 1970s (the PET, Apple, and Tandy) had incompatible disk operating systems, and independent software producers had to opt for one format or the other. Code had to be completely rewritten to make a software product work on a machine other than the one it had been designed for. But CP/M changed all that: its considerable popularity meant that a majority of manufacturers began to adopt it, thus creating a de facto 'standard'. Many computer manufacturers who had chosen the Intel 8080 or Zilog Z80 processors for their machines specified CP/M because it offered a simple way of handling access to the screen, printer, disks, keyboard and so on. And as its popularity increased, more and more CP/M software became available, providing an even greater incentive to adopt it.

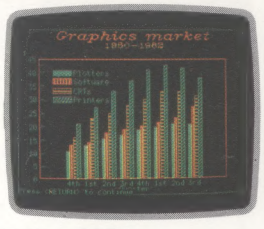
The Control Program for Microprocessors was at first licensed to a few select users. The now famous abbreviation initially stood for 'Control Program/Monitor', but this rather humble title was soon changed! By 1976, Kildall was overwhelmed by requests for the product. He resigned as professor of computer science at a naval college in Monterey and founded Digital Research at Pacific Grove, California.

While CP/M was growing, Digital Research turned its attentions to the multiple-user systems and produced MP/M. This was intended to be compatible with CP/M in every respect, though in its early versions it had none of CP/M's success. Partitioning of the user areas, and other configurations that a systems programmer might need to do, were by no means straightforward, and in some cases file handling differed from CP/M's. However, since the physical costs of microprocessors have dropped as production has risen, the need for several users to share one processor has ceased to make economic sense, and the now-revised MP/M has not proved popular.

Digital Research raised finance from several venture capital companies in 1981, to become a true multinational, with a notably strong presence



LOGO Designs
Digital Research has moved into languages and leads the field with its DR LOGO. Like all good LOGOs, graphics are one of its strong points



IAN McKINNELL

Business Graphics
GSX is a pioneering software package designed to make graphics applications portable between different machines, such as the business graphics package shown here

COURTESY OF DIGITAL RESEARCH INC

John Rowley, President of Digital Research Incorporated

Gary Kildall

Digital Research Incorporated, Massachusetts, USA

COURTESY OF DIGITAL RESEARCH INC

in Europe (where it has offices in the UK, Germany and France). At about the same time, Digital Research was one of the front runners in securing the contract to develop an operating system for IBM's newly-designed Personal Computer. Although Microsoft eventually won the IBM-PC contract, Digital Research was far from beaten. It has since updated CP/M for the Intel 8088/8086 processors in such a way as to make it look very similar to MS-DOS, and it has also gone a step beyond with Concurrent CP/M.

Concurrent CP/M is the converse of MP/M, allowing several different programs to run simultaneously. With this program, a user may work on three different jobs at one time—say, spreadsheet, report generation and electronic mail—switching between them at will. Existing versions of Concurrent CP/M can display each screen—or a portion of it—simultaneously, using the 'windows' feature. New versions of Concurrent CP/M promise to run directly most of the programs written for the IBM PC-DOS.

Among the strategic decisions that Digital Research and many other systems and language houses have taken, is that of moving all its development work into the C language, which is especially notable for its portability. Code written in C need only be recompiled for use on another processor, though this feature has led to accusations of cumbersome coding. It is better, its detractors argue, to do a proper job in assembler for each individual processor. However, it has become increasingly popular, and since the widely used UNIX operating system is itself written in C, the trend towards this language seems irreversible.

Digital Research has certainly been consistent in its view that true portability is only possible through high-level languages. It now provides a variety of languages for a wide range of micros. At the lowest end of the market, however, Digital Research has set up a Consumer Products Division that will sell Personal BASIC, Personal CP/M and its own version of LOGO. Personal CP/M, like CP/M-86, is designed to be stored in ROM, and will soon be available on a Z80 chip by agreement with Zilog. Digital Research describes this as 'microware', and it is sure to prolong the active life of the vast number of ageing 'standard' CP/M programs by making them cheap enough for the home computer user.

Further developments of considerable promise are VIP and GSX. VIP is a cheap visual 'shell' that allows program developers to present a uniform interface to the user, independent of the applications package being executed. Several applications may use the same data, and data can be transferred from one to another. In this respect, VIP is similar to Apple's Lisa and Macintosh technology, but far less demanding of memory. VIP will run in any computer with more than 50 Kbytes of RAM and equipped with 150 Kbytes or more of disk space.

GSX is supposed to do for graphics what CP/M did for disks. It uses a standard set of graphics functions that can be used on a variety of different pieces of hardware. A GSX program will run on a colour screen, a black-and-white screen, a dot matrix printer and a plotter, without any changes. However, Digital Research is experiencing difficulty in creating a new graphics standard, because the system does not produce the same quality as programs that are written specially for one machine. Its popularity has also suffered because of a lack of software.

Digital Research has established itself as one of the major software houses in the microcomputer business. It is not, however, resting on its laurels. After its venture into products such as GSX, VIP and LOGO, there is potential to follow companies like Microsoft into the applications software field. With the unchallengeable background of CP/M's success, the company looks set for a long future.

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 1 TO 12

A

Absolute jumps 219
Access time 13
Accountant **121-122, 152-153**
Accumulator 13, 137
Accumulator register 116
Acorn **60**
Acoustic couplers 13, 101
Acronym 13
Active electrical components 138
ADA 13
A-D converter 28, 223
Adder 28
Adding machine program 170
Adding records 227
Address 28
Address bus 135
ADSR 28
AIM 65, **109**
Alexander, Nick 180
ALGOL 49
Algorithm 49
Allophones 49
Alphabetical index program 205
Alphanumeric 49
Amending a record 227
Analogue signals 223
AND **8, 32, 46, 66,** 145
Ant Attack **6**
Apple computers **163**
Arithmetic logic unit 137
Array 88
Artificial intelligence 88
ASCII 57, 108
ASCII Microsoft 142
Assembler 108
Assembler directives **156**
Assembly language **108, 116**
subroutine calls **236**
pseudo-opcodes **156,** 236
Asynchronous transmission 108
Atari **39-40**
disk commands **64-65**
600XL/800XL **189-191**
810 disk drive **63-64**
1027 printer 190
Attribute 108

B

Background processing 129
Backplane 129
Backup 129
Bandwidth 30, 148
Bank switching 148
Bar codes 148
BASIC 20, 168
BBC
BASIC **54-55**
disk **84**
disk commands 85
Micro and Viewdata 133
BCD 168, 195
BDOS 182
Benchmark 168
Binary-coded decimal system 168, 195
Binary conversion programs 38
files 185
number system **36-37**
BIOS 182
Bistable 168
Bit 36, 188
Bit-copier 188
Bit-error 188
Bit-manipulation 188
Bit-mapped 188
Bit-twiddling 188
Block 188
Block menu 27
Book-keeping packages **152**
Boole, George 32, 126
Boolean algebra **8, 32, 46, 106, 126**
Bootstrap 188
Branson, Richard 180
Breadboards 144, 188
Break 208
British Telecom 101
Bubble memory 208, 210
Bubble sort 208
Buffers 208
Bus 208
Bushnell, Nolan 39
Byte 36

C

C language 240
CAD (see computer aided design)
CAL (see computer aided learning)
CALL 235, 238
CAM (see computer aided manufacturing)
Capacitors 115, 139, 194
Carry **176**
Cash Trader 152
Cassette-based business programs 113
Cassette tape 4
CBM
700 120
8032 120
9000 120
PET 119
CCP 182
Ceefax 132
Character set 19
Chip development **162**
Circuit *design* 106
tester 86
CNC 235
Colour codes and resistors 138
Command control program 182
Commodore BASIC 94
Commodore Business Machines **119**
Commodore-64 **10-12,** 120
disk commands **53**
DOS **52**
graphics **214, 232**
memory map 12
Commodore SX-64 10
Commodore Vic-20 120
Computer aided design 171, 235
Computer aided instruction 235
Computer aided learning 235
Computer aided manufacturing 235
Computer-controlled devices 221
Concurrent CP/M 240
Control bus 135
Cosine function **174**
CP/M 183, 239-240
CPU 43, 148
history **161**
internal organisation **136**
micro-operations 157
Current 114

D

D-A converter 28, 223
Data bus 135
de Morgan's law **46-48**
Decoders **146-147**
DFS 84
Digital Research **239-240**
Diodes 138
Disk
density 124
drives **4-5, 63-64, 84**
filing system 84
operating systems 5, **181-183**
sectoring **124-125**
Display memory 117
DOS 5
Dot crawl 29
Double precision 175
Dragon Data's Stock Recording System 172, **192, 212-213**
Dragon disk
commands **105**
DOS **104**
disk unit 131
Dragon 32 131
Dragon 64 **130-131**

E

Educational software **21-23**
Electronic mail 101
Encoders **146-147**
EEPROM 4
Ethernet system 100
Exam revision packages **61**
Expert system **81-83**

F

Faggin, Frederico 220
Feedback 221
Fields 226
File handling **184-185, 226-227**
Flip-flops 168
Floppy disk 5
Floppy tape 34
Formatting disks **124-125**
Forsyth, Richard 81
Full adder 165

G

Games Designer **41-42**
Games generator packages **41-42**
Gates, Bill 20
Grafpad **169-171**
Graphic design **224-225**
Graphics generators **132-134**
tablets 169
GSX 240
GTIA chip 190

H

Half adder circuits **164-165**
Hard-wired modems 101
Help menu 26
Hexadecimal convertor 99
number system **56-58**
Hinkley, Norton 199
HULK **83**

I

IBM 89
IBM PC 89-91
IBM PC software 90
Index & general purpose registers 137
Indexed addressing 196-197
Indirect addressing 197-198
Inference engine 82
Information provider (Prestel) 132-133
Integer variables 175
Integrated software 159
Integration 172
Intel 8008 161-163
8080 161-163
8086 161-163
8088 161-163
Interactive approach 112

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 1 TO 12

Interactive video 203
IP (Prestel) 132

J

J-K flip-flop 168

K

Karnaugh maps 92-93, 126
Kildall, Gary 161, 239
Knowledge-acquisition module 82
Knowledge-based systems **81-83**

L

LAN 100
Languages 2
Large scale integration 194
Laser disks **201-203**
Latching switches 115
Light emitting diodes 115
Loading the accumulator 116
Locus 224
Logic **8-9, 32-33, 46-48, 66-67, 92-93, 106-107, 126-127, 144-145, 146-147, 164-165, 166-167, 186-187, 194-195, 206-207, 228-229**
LO-HI 96
Loops **216-219**
Lovelace, Ada 13
LSI 194

M

Machine code **16-19, 36-38, 56-58, 76-78, 96-99, 116-118, 135-137, 156-158, 176-179, 196-198, 216-219, 236-238**
Mask 232
MATE 66
Medium scale integration 194
Memory maps **58**
Microcomputers in education **21-23**
Microdrive **34-35**
Microledger 121, 152
Micronet 800 **101-103**
Microprocessors 43, 136, 148
history **161**
Microsoft **20**
Modem 13, 40, 102-103
Monitor **29-31**
Monitor program (Spectrum, BBC, Commodore 64) 118
MOS Technology 163
6502, instruction set 163
Motorola 43, 161
6800 43,161
*6809*43, 161
68000 series 43, 161
MP/M 239-240
MSI 194
MSX BASIC 142
MSX Standard **141-143,** 149-151
Multimeter **86-87**
Multiplexing 207
MUPID terminal 134

N

NAND gates **166-167**
Non-latching switches 115
NOR gates **166-167,** 228-229
NOT 8-9, 32-33, 46, 146
NTSC 30-31

O

Object code 108
Ohm, Georg 86, 114
Ohm's Law 114, 138
Omicron's Powerstock package 173
Op-code **98-99**
Operating systems **181-183,** 239
OR 8-9, 32-33, 46, 66, 145
Oracle 132
Oric-1 140
Oric Products International 140
OS9 131

P

Paged memory **36-37**
PAL 30-31
Parity bit generators **186-187**
Passive electrical components 138
Peddle, Chuck 119
PEEK 232-233
Persistence 31
PIPS 159
POKE 232-233
Power Law 114
Prestel 101, 132
Priority encoder **186-187**
Prism 101
Processor status register 137, **176-177**
Program Counter 137
PROM 4
Pseudo-op **156, 236**
Pulsar's Stock Control System 173

Q

Quick command menu 27
Quick-Count's Book-keeping System 121
Quicksilva 6
Quill 42

R

Radian measure 174
Radix 148
Records 226
Registers 116, **229**
Relative jumps 219
Relays 222
Resistance 114
Resistors 115, **139**
Retrieving records 227
RGB 30
Rockwell's AIM **109-111**
Romox Corporation 39
R-S flip-flops 228-229

S

S-100 system/bus 163
SECAM 30-31
Sector data block 125
Sector header 125
Sequential circuits **228-229**
Serial files **204-206, 226-227**
Servo motors 223
Seven-segment displays **206-207**
Sharp PC-5000 **210-211**
Sharp thermal printer 210
Shiina, Takayoshii 159
Shima, Masatoshi 220
Short addressing 196
Simplified XOR gate 47
Sinclair Microdrive **34-35**
Sinclair Spectrum **50-51**
Sinclair Spectrum word processor 51
Sine function **174-175**
Single-byte arithmetic 177
Single precision 175
Software portability 183
Soldering **44-45**
SORD **159-160**
Source code 108
Speakers 115
Spectravideo 318 **149-151**
Spectravideo 328 150
Spectrum BASIC **14-15, 24-25**
Stack Light Rifle **230-231**
Stack pointer 137
Start bit 108
Stepper motors 223
Stock control **172-173, 192-193, 212-213**
Stop bit 108
Storing machine code 117, **135-137**
Stringy floppy 34
Subhunter program **214-215, 234**
SuperPET 120
Synchronisation pulses 29
Synchronous transmission 108

T

Tandy Corporation **199-200**
Tandy, Charles 199
Tandy, David 199
Teletext systems 132-133
Thermal printers **70-72**
Tramiel, Jack 119
Transistors 115, **139, 144,** 194
Trigonometric functions **154-155, 174-175**
Truth table 8, 126
TTL chips **194**
Turing, Alan **88**

U

User interface 82
User port 222

V

Variable resistors 115
Venn diagrams **46,** 126
Vic-20 120
Viewdata graphics **133**
Viewtext 133
VIP 240
Virgin Games **180**
VisiCalc 163
VLSI 144, 194
Voltage 114
VTX 5000 102

W

Word processing **26-27**
WordStar **26-27**
Wozniak, Steve 163

X

Xerox PARC **100**
XOR gate **47, 186-187**

Z

Zero page addressing **196**
Zilog 161-163, **220**
Z80 **161-163, 220**
Z80 instruction set **179, 238**
Z800 162-163, 220
Z8000 162-163, 220
ZX Interface 1 34-35

6502 microprocessor 163
6502 instruction set **179, 238**
6800 microprocessor 43, 161
6809 microprocessor 43, 161
68000 series 43, 161
8008 microprocessor 161-163
8080 microprocessor 161-163
8086 microprocessor 161-163
8088 microprocessor 161-163